博 士 学 位 论 文

面向多核系统的实时调度算法研究

导　师：于　戈　教授

研究生：关　楠

东北大學

二０一二年八月

分类号_____密　级_____

U D C _____

学　位　论　文

面向多核系统的实时调度算法研究

作 者 姓 名：　关　楠

指 导 教 师：　于　戈　教授

申请学位级别：博　士　　学 科 类 别 ：　工　学

学科专业名称：嵌入式系统及应用

论文提交日期：2012 年 8 月 10 日　论文答辩日期：　2012 年　月　日

学位授予日期：　　答辩委员会主席：

评　阅　人：

東北大學

2012 年 8 月

A Dissertation for the Degree of Doctor in Embedded Systems and Applications

Research on Real-Time Scheduling for Multi-core Systems

by GUAN Nan

Supervisor: Professor YU Ge

Northeastern University

August 2012

独创性声明

本人声明，所呈交的学位论文是在导师的指导下完成的。论文中取得的研究成果除加以标注和致谢的地方外，不包含其他人已经发表或撰写过的研究成果，也不包括本人为获得其他学位而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

学位论文作者签名：

日　期　：

学位论文版权使用授权书

本学位论文作者和指导教师完全了解东北大学有关保留、使用学位论文的规定：即学校有权保留并向国家有关部门或机构送交论文的复印件和磁盘，允许论文被查阅和借阅。本人同意东北大学可以将学位论文的全部或部分内容编入有关数据库进行检索、交流。

作者和导师同意网上交流的时间为作者获得学位后：

半年 □　　一年 □　　一年半 □　　两年 □

学位论文作者签名：　　导师签名：

签字日期：　　签字日期：

面向多核系统的实时调度算法研究

摘 要

随着多核处理器的飞速发展，越来越多的嵌入式实时系统设计者选择多核处理器作为硬件平台，以满足各类应用不断增长的高性能与低功耗的需求。更高质量和更高效率的多核处理器实时调度技术已成为这一发展趋势的迫切需要。在过去的四十年中，面向单核处理器的实时调度技术已经发展得比较成熟。相比之下，面向多核并行体系结构的实时调度，无论从理论方面还是系统实现方面依然面临着巨大挑战。

基于这一背景，本文研究面向多核体系结构的实时调度技术，旨在突破多处理机调度中的重要基本理论难题，并解决在多核平台上构建实时系统的实际挑战。多处理机调度主要分为全局调度和划分（及准划分）调度。本文分别针对这两类调度算法中的关键理论问题给出了新的理论结果（包括全局调度的关键时刻、全局调度的有限相应时间条件、准划分调度中的资源利用率界限），并提出了一系列调度技术来提高多处理机系统的平均实时性能（包括不可抢占全局调度技术、任务实例级别优先级分配技术、准划分调度中基于响应时间的划分技术、准划分调度中参数化的资源利用率界限）。主要贡献点概括如下：

（1）建立了多处理机全局调度关键时刻的概念，在此基础上提出了一种针对可抢占全局固定优先级调度的响应时间分析新技术，在不牺牲分析效率的前提下大幅度提高了分析的精确度。在这一理论基础之上，建立了可抢占全局固定优先级调度下任务具有有限响应时间的一般性条件。

（2）提出了一种新的针对不可抢占全局固定优先级调度的可调度性分析技术，并结合大量的模拟实验，推翻了从单处理机实时调度中衍生出来并被普遍接受的关于“可抢占调度的实时性能总是好于不可抢占调度”的错误观念，并对这种现象进行了深入分析，系统分析了如何利用不可抢占调度来提高系统的实时性能。

（3）提出了一种固定实例优先级的全局调度算法及相应的分析技术。该算法结合了固定任务优先级分配与最早截止期优先（EDF）两类算法的优点，通过发掘任务实例之间的优先级顺序来大幅度提高系统的可调度性。该算法在设计阶段仅对有限个具体的任务实例进行优先级分配，以构建运行时系统的负载抽象表达；在运行时，通过复用上述优先级分配方案实现高效的在线调度。

（4）提出了一种新的准划分调度算法，将单处理机调度中著名的Liu&Layland资源利用率界限推广到多处理机调度模型，解决了实时调度领域一个近四十年悬而未决的重

要理论问题。其基本思想是使用与装箱问题中的“最坏适用递减”启发式算法类似的任务划分顺序，来使任务切割只发生在高优先级任务中，并利用高优先级任务具有较大松弛时间的特性，来抵消任务切割所带来的负载增长效应。

（5）提出了一种新的准划分调度算法，将单处理调度中大部分的参数化资源利用率界限扩展到了多处理机调度。此外，该算法由于使用响应时间分析来决定一个处理器上可以接纳的最大负载，因此获得了比现有算法更好的平均情况实时性能。

此外，本文还研究了面向共享缓存的多核实时调度技术。多核处理器的一个全新特性是存在大量的片上共享硬件资源（如共享缓存等），对共享资源的并行访问使得一个任务的执行时间依赖于其它同时执行的任务，因此动摇了传统实时调度中“已知任务的最坏情况执行时间”这一基本假设，为多核实时调度及可调度性分析带来了前所未有的挑战。本文提出了一种全新的共享缓存敏感的多核实时调度及可调度性分析技术。该方法首先通过划分共享缓存来减少任务间干涉从而提高了系统的时间可预测性，在此基础上，从“处理机”和“缓存”两个维度上解决资源分配与调度的问题，并相应的给出了两种可调度性判定的方法。

综上，本文研究了面向多核处理器的实时调度问题，内容涵盖了多个调度算法种类（全局与划分，可抢占与不可抢占，固定任务优先级与固定实例优先级）。本文的研究成果为面向多核处理器的实时系统的设计与分析提供了重要的理论依据，并对解决在多核系统上部署实时系统的实际问题具有很好的参考价值。

关键词：实时系统；多核处理器；多处理机调度；可调度性分析；响应时间分析；资源利用率界限；共享缓存

Research on Real-Time Scheduling for Multi-core Systems

Abstract

The rapid development of multi-core processors leads to a constantly increasing trend of deploying real-time systems on multi-core platforms, to satisfy the dramatically increasing high-performance and low-power requirements. This trend demands effective and efficient multiprocessor real-time scheduling techniques. The uniprocessor scheduling problem has been well studied during the last 40 years. However the multiprocessor scheduling problem to schedule tasks onto parallel architectures is a much harder challenge.

This thesis studied the design and analysis of real-time scheduling algorithms for multi-core processor systems. The major target of this thesis is to solve several fundamental theoretical problems in the multiprocessor scheduling model, as well as to consider various aspects in constructing practical real-time systems on multi-core processors. Multiprocessor scheduling is usually categorized into two paradigms: global scheduling and (semi-) partitioned scheduling. This thesis contributes to both with new fundamental results (including the critical instant in global scheduling, conditions for bounded response time in global scheduling, and utilization bound in semi-partitioned scheduling), as well as proposing several techniques to improve the average-case real-time performance in multiprocessor scheduling (including non-preemptive global scheduling, job-level priority assignment in global scheduling, semi-partitioned scheduling based on response time analysis and parametric utilization bounds in semi-partitioned scheduling). The main contribution of this thesis can be summarized as follows:

(1) This thesis proposed a novel response time analysis technique for preemptive global fixed-priority scheduling, based on a concept similar to the well-known critical instant in single-processor scheduling. This new technique can significantly improve the analysis accuracy without degrade the efficiency. Futher, a general condition guaranteeing the bounded response time of tasks in global fixed-priority scheduling is established based on the above technique.

(2) This thesis proposed a novel schedulability analysis technique for non-preemptive global fixed-priority scheduling, and conducts empirical simulation experiments, which disclose that in multiprocessor scheduling non-preemptive scheduling can outperform

preemptive scheduling in many cases. This counters the widely accepted knowledge originating from single-processor scheduling that preemptive scheduling is superior to non-preemptive scheduling. The thesis also dicusses under what kind of circumstance non-preemptive can greatly improve the system real-time performance.

(3) This thesis proposed a novel job-level fixed-priority scheduling algorithm and its schedulability analysis techniques. The proposed algorithm can greatly improve the system real-time performance by exploring the priority order among jobs, and can be viewed as a combination of the advantage of both fixed-priority scheduling and EDF scheduling. This algorithm offline constructs the abstract system workload and only assign priority to a limited number of jobs, by which the runtime scheduler can dispatch job priorities and guarantee the timing correctness of the system.

(4) This thesis proposed a novel semi-partitioned scheduling algorithm, which generalize the famous Liu&Layland utilization bound in single-processor scheduling to multiprocessor scheduling. This algorithm is based on a partitioning approach similar to the "worst-fit decreasing" heuristics in bin-packing problem, such that task splitting only happens with high priority tasks, which solves the workload increase caused by task splitting. This algorithm solves a 40-years long-standing open problem in real-time scheduling.

(5) This thesis proposed a novel semi-partitioned scheduling algorithm, which can generalize most parametric utilization bounds in single-processor scheduling to multiprocessor scheduling. This algorithm uses response time analysis, instead of the utilization bound test, to decide the maximal workload that can be assigned to a processor, and thus it has a much better average-case real-time performance then the above algorithm.

Moreover, this thesis also studied cache-aware real-time scheduling on multi-cores. An important new feature of multi-core processors is using many on-chip shared resources (like shared cache). The timing behavior of a task depends on other co-running tasks due to the accesses to the shared resources, which invalidate a basic assumption in traditional real-time scheduling research that the worst-case execution time of each individual task are known. This thesis proposed a novel cache-aware real-time scheduling technique, which guarantee the system timing precictability by avoiding inter-task shared cache contention, and presented sufficient schedulability test conditions for the novel scheduling which require both processors and cache blocks as processing resources.

In summary, this thesis studied various real-time scheduling algorithms for multi-core

systems, covering different scheduling paradigms (global and partitioned, preemptive and non-preemptive, fixed-task priority and fixed-job priority). The results of this thesis serve as theoretical foundations as well as provides practical insights for the design and analysis of real-time systems on multi-cores.

Keywords: real-time systems; multi-core processor; multiprocessor scheduling; schedulability analysis; response time analysis; utilization bound; shared cache

目 录

第1章 绪 论

1.1 研究背景及意义

实时系统[1]是指那些需要对外部产生的输入在特定的时间内做出响应的信息处理系统；这些系统的正确性不但取决于其逻辑结果，还取决于产生结果的时间。实时系统广泛存在于各种各样的嵌入式系统（Embedded Systems）[2]与 CPS（Cyber-Physical Systems）[3]应用之中，随着信息技术与人类生活的融合不断加深，实时系统已经成为计算机系统发展中至关重要的方向之一。

根据对时间要求紧迫程度的不同，可以将实时系统划分为硬实时和软实时两种[4]。硬实时系统不允许任何不满足时间特性的情况出现，一旦出现将会导致灾难性的后果。通常关键任务系统（Critical Systems）都是硬实时系统，例如汽车控制系统、航空控制系统、复杂医疗系统以及武器系统等。与硬实时系统对应的是软实时系统，在软实时系统中，即使时间要求在一定范围内不被满足，系统仍能够正常运行，但是系统所提供的服务质量（QoS）则会降低。多媒体系统就是一种典型的软实时系统，例如在视频播放的过程中，少数几帧的丢失或延迟通常可以被容忍且不会导致系统瘫痪，但是却会降低观众的欣赏质量。

由于实时系统，尤其是硬实时系统，对系统的时间行为特性有严格的要求，因此在实时系统的设计中，系统时间可预测性（Timing Predictability）是设计者需要考虑的首要问题。系统的可预测性涉及到两方面问题：其一是，系统自身是如何运行的；其二是，如何对系统的运行情况进行预测与分析。

更确切地说，第一个问题是如何安排系统中各个任务在何时执行。嵌入式系统通常是多任务系统，即多个任务共享同一组系统硬件资源。系统需要在运行时决定在何种情况下运行哪个任务，以期满足任务的时间特性要求。这部分功能通常与如操作系统（或中间件等其他运行时系统）中的调度算法（Scheduling Algorithm）有关。调度算法有许多种。比如，有些调度算法的主要目标是保证系统的公平性，即保证每个任务都能轮流得到执行；而在实时系统中，系统设计的首要目标是保证每个任务满足各自的时间特性要求。对于实时系统而言，衡量一个调度算法好与坏的标准是看其能够多大程度地满足这一要求。总而言之，实时系统设计中，设计者面临的第一个基本问题是如何设计一个“好”的调度算法，尽可能使所有的任务满足各自的时间特性要求。

但是，对于保证系统的可预测性而言，仅仅设计一个“好”的调度算法还是远远不

够的，更重要的是如何对系统的运行情况进行预测与分析，即判定系统在该调度算法下是否一定满足所有的时间特性要求。由于实时系统，尤其是硬实时系统，对于执行时间的要求非常苛刻，因此通常需要在系统实际运行之前对系统的时间特性进行验证，以确保系统在运行过程中规定的时间特性要求得到满足。具体来说，就是在一个给定的调度算法之下，判断每个任务能否在指定的时间点（即截止期，Deadline）之前完成，通常称这种分析为可调度性分析（Schedulability Analysis）[5]。如前文所述，嵌入式系统一般都是多任务系统，而且任务一般是复发的（Recurring），即周而复始无限重复执行的，因此，对系统的运行情况做出预测与分析通常非常困难。有些时候，为了使对系统的分析能够在有限的时间和资源约束下完成，设计者不得不某种程度的牺牲分析的精确性而进行保守近似（Overapproximation），即给出判定系统的时间特性要求是否满足的充分而非必要条件。总之，对一个给定的实时调度算法进行可调度性分析，是实时系统设计者们所面临的另一个重要问题。

现代计算机系统发展的另一个主要趋势是“多核（Multi-core）处理器逐渐取代单核（Single-core）处理器，成为包括嵌入式系统在内绝大部分计算机系统的硬件基础”。在单核处理器时代，处理器芯片制造厂商主要是通过不断提高芯片频率来获得更高的处理性能。Intel 这一全球最大的芯片制造厂商，在启动其 Pentium 4 芯片的研发时，曾期望在单核处理器上获得 10GHz 的频率。但实际上这一目标远未实现。其根本原因是，处理器的功耗随着频率的提高呈二次函数增长，因此为了获取更快的频率，芯片的功耗变得极高，并最终超越计算机可以负担的极限。2004 年 5 月 17 日，Intel 取消了其下一代单核处理器 Tejas 的研发计划。2006 年 7 月 27 日，Intel 正式发布了第一款双核处理器（Core Duo 系列处理器），标志着个人计算机系统迈入了多核时代。如今，4～8 核处理器已经成为个人桌面电脑的标准配置。2009 年 10 月 27 日，多核处理器厂商 Tilera 宣布推出全球第一款核心数量多达 100 个的微处理器——“TILE-Gx100”。与个人桌面电脑相比，嵌入式系统对于功耗的要求通常更为苛刻，因此更需要依赖于多核处理器来提高单位功耗下的运算能力。目前世界上主要的嵌入式处理器厂商都已推出了多核嵌入式处理芯片，如 ARM 的 ARM11 MPCore[6]和 Cortex-A9 MPCore[7]，TI 的 OMAP4[8]与 DaVinci[9]，Freescale 的 PowerQUICC[10]，Sony、IBM、Toshiba 联合推出的 Cell[11]，NXP 的 Nexperia[12]，STM 的 Nomadik[13]，等等。

多核处理器在嵌入式系统中的应用与普及为实时系统的设计与分析带来了巨大挑战。其中最核心的问题是：如何设计面向多核系统的实时调度技术，从而在保证系统行为的时间正确性的同时，充分利用多核处理器所提供的强大计算能力。多核实时调度问题的出现，大大拓展了传统的实时调度研究，因此在近年来成为了学术界的主要研究热

点之一。自上世纪 60 年代末实时调度问题出现，经过学术界近 40 年的努力，面向单处理机系统的实时调度技术已经趋于成熟，并出现了一系列重要的理论成果。然而单处理机模型下的绝大多数理论成果无法应用于多核处理器模型。例如，对于可抢占周期性任务，EDF（最早截止期优先，Earliest Deadline First）调度算法是单处理机模型上的最优算法，其可以达到 100%的资源利用率界限（Utilization Bound）[14]，但是 EDF 在多核处理器模型上非但不是最优，而且会导致极低的资源利用率界限 [15]。类似例子还有许多，其中一些将在下文进一步叙述。总而言之，在单核处理器模型下积累的实时调度理论成果，无法有效地指导基于多核平台的实时系统的设计，面向多核系统的实时调度中的许多基本理论问题尚未得到解答。这一现状已经成为了多核处理器在嵌入式系统中发挥优势的主要瓶颈之一。因此，本文将结合现有研究成果和基础，围绕多核实时调度算法的设计与分析开展研究工作，重点解决各种类型的多处理机调度算法中的重要基本理论问题，以及在多核处理器上构建实时系统的实际挑战。

1.2 国内外研究现状

多核处理器上的实时调度问题的研究起源于上世纪 60 年代末 70 年代初以多处理机系统（一个系统由多个处理器芯片组成）为背景的研究工作。因此在学术界人们也通常称这个问题为“多处理机实时调度”。

1969 年 Liu 等学者指出，多处理机实时调度是一个比单处理机实时调度困难许多的问题[15]。在文中他指出：“单处理机调度的结果中几乎没有可以直接扩展到多处理机的情况。增加了处理器的个数实际为调度问题增加了一个维度。一个任务只能同时使用一个处理器，这样一个简单的事实为多处理调度带来了惊人的难度”。

大概十年之后，Dhall 和 Liu 发表了一篇影响了多核实时调度研究近二十年的论文[16]。论文指出，如果允许一个任务在不同的处理器之间迁移，那么使用单处理机上的最优调度算法 RMS（Rate Monotonic Scheduling）及 EDF，会导致任意低负载的任务集不可调度。这样一个令人意外的结论表明，虽然直观上允许任务在不同的处理器之间自由迁移会使系统负载更加平衡从而提高系统的实时性能，但是在最坏情况下这可能会导致非常低的系统性能。这个现象被称为 Dhall 效应。

由于 Dhall 效应的负面结论，在此后的二十年间，研究者们基本放弃了对允许任务迁移的调度算法的研究，而将精力集中在如何将一个任务系统划分成若干个子系统，并在每个处理器上单独执行一个子系统的方法。这类问题与装箱问题（Bin Packing）非常类似，一般情况下具有 NP-hard 复杂度。研究者们提出了许多与装箱问题解法类似的近似算法。在研究领域，通常采用不同的指标来描述一个近似算法的优劣。其中一类工作

考虑如何得到与最优算法近似度最高的算法[15, 17~19]，这类工作与计算理论研究中相关工作的方法类似，通过证明一个算法的近似率（Approximation Ratio）来表明该算法的优劣。另一类工作考虑如何计算某种算法的资源利用率界限（Utilization Bound）[19~25]。资源利用率界限不但可以用来表明一个算法的性能优劣，还可被直接用来验证一个任务系统是否可以被调度。因此，上述第二类工作的结果在实时调度领域的应用意义相对更大。由于装箱问题本身固有的限制，任何不允许任务迁移的多处理机调度算法的利用率界限都无法严格超过 50%。

研究者们对于可迁移调度算法的忽视一直持续到 Phillips 等人发表论文[26]。此论文指出，Dhall 效应的由于高负载的任务被错误地赋予了低优先权。在单处理机调度中，任务的负载对于决定调度的优先级没有意义，而只有任务的紧急程度才对优先级有意义。如果将这样的做法推广到允许迁移的多处理机调度，则会产生 Dhall 效应。Phillips 等学者的研究结果的意义在于，指出了允许迁移的多处理机调度中另一个重要的任务特性指标——负载。通过调整对高负载任务的处理方式，或者获得关于系统中高负载任务的详细信息，可迁移多处理机调度有望获得更好的实时性能。在此论文发表后的十年间，出现了一大批工作研究可迁移多处理机调度。在可迁移多处理机调度中，有两类主要的方法，一是全局调度（Global Scheduling），二是准划分调度（Semi-Partitioned Scheduling）。前者允许所有任务可以在任何一个空闲处理器上执行，后者仅允许一小部分任务进行迁移（其它任务不可迁移）。

对于全局调度，问题的难点在于无法对算法的可调度性进行精确的分析。具体的讲，在此类调度中，一般情况下无法找到系统的“关键时刻”（Critical Instant）。关键时刻是指调度中一个特定的任务释放模式，系统如果在这个释放模式下可被调度，则在任何任务释放模式下都保证可以被调度。换句话说，关键时刻就是一个具体的系统最坏情况，因此对于系统的分析只需要考虑这一个具体的情况。关键时刻是单处理上实时调度分析的基础，诸如响应时间分析和资源利用率界限等方法都是基于关键时刻开展分析的。在多处理机上，由于无法找到关键时刻，也就说无法找到一个具体的情况来代表系统的最坏行为，导致可调度性分析变得困难得多。对这个问题的研究比较活跃的研究者包括 Baruah、Baker、Andersson 以及 Bertogna 等人，他们的工作重点考虑如何构造出近似的系统任务负载状况，以在没有关键时刻的情况下对可调度性进行判断[27~43]。虽然这些工作取得了一定进展，但是总的来说这些全局调度的分析技术还很不精确。此外，他们的工作普遍采用 RMS 及 EDF 这些单处理机调度的最优算法作为全局调度的基础，然而这些算法在多处理机全局调度中的性能往往非常不理想。

对于准划分调度，问题的难点在于如何划分任务系统。如前所述，划分调度问题与

装箱问题非常相似，即考虑如何将一些不同尺寸的物品（任务）放置到若干个容器（处理器）中。在准划分调度中，我们允许一些任务在不同处理器上进行迁移。任务在不同处理上的迁移可以理解为把这个任务切割（Split）成若干个更小的任务。在装箱问题中如果允许对物品进行切割，问题则变得非常容易。但是，在多处理机调度问题中，允许任务切割并不能直接简化问题。其主要原因是，由于被切割的子任务间存在逻辑上的先后执行关系，在截止期的限制下一个任务被切割后未必会变成两个“更小”的任务。对于这个问题的研究比较活跃的研究者包括 Andersson、Kato、Lakshmanan 等人[44~54]，他们的工作重点考虑如何通过任务切割来提高系统的利用率界限。例如，Lakshmanan 等人设计了一个具有 65%资源利用率界限的固定优先级准划分调度算法[54]。但是这个资源利用率界限仍低于单处理调度中著名的 Liu&Layland 资源利用率界限 $N(2^{1/N}-1)$，即仍无法到达和单处理机调度中最优调度相同的实时性能。

除了上述多处理机调度中的基本理论问题外，随着多核处理器的广泛应用，研究者们也越来越多地关注多核处理器中新的体系结构特性对实时调度的影响。在传统的实时调度研究中，人们通常假设每个任务的最坏情况执行时间（Worst-Case Execution Time, WCET）是已知的，然后根据任务 WCET 以及其它参数来设计调度算法或进行可调度性分析。对于单处理器系统，或者由多个单处理器组成的多处理器系统中，这个假设是切合实际的，而且已经存在相对成熟的技术来获取每个任务的 WCET[55]。但是在多核系统中，这样的假设实际与否却存在着疑问。其主要原因是，在多核处理器上，各个核之间需要细粒度地共享许多资源，因此一个任务的执行时间会受到其它核上程序的影响。比如，多核系统中对任务执行时间影响最大的共享资源之一是共享缓存（Shared Cache）。在运行时，任务 A 的某个共享缓存内容 X 可能会被其它核上运行的任务置换出去，因此下一次任务 A 访问 X 时会造成缓存缺失，进而延长其执行时间。但是，X 是否会被置换出缓存取决于什么任务和 A 同时运行。也就是说，在计算任务 A 的 WCET 时，我们需要知道有关调度的详细信息。这一现象动摇了传统实时调度研究中一个默认的基本假设：WCET 可以在进行调度分析之前预先获得。这对于实时调度的理论研究来说是一个重大的问题，因为这意味着在过去近四十年中所有实时调度理论的研究成果有可能都不再适用于多核系统。近年来，一些研究者提出了一些方法来解决多核处理器上共享资源对实时任务的影响问题。其中一些研究者重点考虑如何将由共享总线和存储器接口冲突造成的影响计入到传统的实时调度分析体系中[56~61]。共享总线的行为相对比较简单，因此对其进行分析相对比较容易。相比之下，共享缓存的行为更加复杂，因此难以结合到传统的实时调度分析体系中。一些工作在多核调度中考虑了共享缓存所带来的影响[62~63]，这些工作只考虑了如何软实时系统中有效地利用共享缓存，而无法对系统的硬实

时性能提供任何保证。

1.3 本文研究内容

本文针对多核实时调度研究领域尚存在的不足，围绕面向多核处理器实时调度算法的设计与分析开展研究工作，重点解决了各种类型的多处理机调度算法中若干重要的基本理论问题，以及在多核处理器上构建实时系统的若干实际挑战。本文在全局调度，准划分调度，以及共享缓存敏感的调度等三方面展开了研究。本文的研究内容主要有以下几个部分：

（1）全局调度算法的研究。主要研究基于固定任务优先级可抢占的与不可抢占全局调度的可调度性分析与响应时间分析问题，并提出了一种新的固定实例优先级全局调度算法及分析技术：

a）本文对于可抢占全局固定优先级调度，提出一种新的可调度性和响应时间分析技术。该分析技术的特点是建立了一个与单处理机调度中的关键时刻类似的概念，通过构造一个相对具体的情形来代表系统的最坏情况行为。通过对这个特定情形进行分析，可以得到系统可调度性和响应时间的安全近似。与现有技术相比，该新技术在不增加分析复杂度的前提下能够大大地提高分析的精确性。本文还研究如何将此技术应用于任务截止期大于任务周期的情况，并建立了可抢占全局固定优先级调度中任务具有有限响应时间的一般性条件。

b）本文对于不可抢占全局固定优先级调度，提出一种新的可调度性和响应时间分析技术。该分析技术的特点是为不可抢占固定优先级全局调度算法建立了一个新的“问题窗口”概念，并将上一部分中针对可抢占调度的分析技术应用于不可抢占调度。此外，本文通过为不可抢占全局调度进行更加精确的分析以及进行大量的模拟实验，发现了固定优先级全局调度中一个重要的现象：在很多情况下，不可抢占调度的实时性能要好于可抢占调度。这个现象推翻了从单处理机实时调度中衍生出来并被普遍接受的概念：就实时性能而言，可抢占调度总是好于不可抢占调度。本文对这种现象进行了深入分析，并讨论了在何种情况下可以更好地利用不可抢占调度来提高系统的实时性能。

c）本文提出一种固定实例优先级的全局调度算法及相应的分析技术。该调度算法与以往的全局调度算法有显著的区别：算法通过发掘任务实例之间优先级顺序来大幅度提高系统的可调度性。这种方法可以看作是将基于固定任务优先级分配技术与 EDF 这两种现有全局调度算法优点的结合。该方法面临的一个重要挑战是，如何为系统动态产生的无限的任务实例分配优先级。本文提出的算法通过构建运行时系统负载的抽象，只需在设计时对有限个具体的任务实例进行优先级分配，而在运行时系统将通过复用上述

优先级分配方案来正确进行调度。实验表明，该算法的性能明显优于基于固定任务优先级分配和 EDF 的全局调度算法。

（2）准划分调度算法的研究。这部分研究工作解决了实时调度领域一个著名的难题，即如何将单处理机调度中的 Liu&Layland 资源利用率界限扩展到多处理机调度模型。本文提出两个全新的固定优先级准划分调度算法。

a）第一组算法首先将著名的 Liu&Layland 资源利用率界限扩展到了多处理机调度。该组算法具有多项式复杂度。其基本思想是使用与装箱问题中的“最坏适用递减（Worst-Fit-Decreasing）”启发式算法类似的任务划分顺序，来使任务切割只发生在高优先级任务中，并利用高优先级任务具有较大松弛时间的特性，来抵消任务切割做带来的负载增长。此问题中的一个难点是如何处理负载很大的任务，即所谓的“重型”任务。本文提出的算法通过设计一个“重型”任务的预先划分过程，来将“重型”任务精确地区分成两类，一类是可能在切割中引起负载增长的任务，另一类则不能引起负载增长。算法对两类不同的“重型”任务采用不同的划分机制，从而解决了任务切割造成的负载增长问题。

b）第二组算法将单处理调度中大部分的参数化资源利用率界限扩展到了多处理机调度。Liu&Layland 资源利用率界限虽然简单，但是很悲观。在系统设计时，如果可以知道任务的更多参数，则有可能使用更高的参数化资源利用率界限来进行可调动行判断。本文提出的第二组准划分调度算法可以将单处理上大部分的参数化资源利用率界限扩展到多处理机调度。该组算法具有伪多项式复杂度。本文提出的解决方法是通用的，也就是说该方法与一个参数化资源利用率界限的具体形式无关。该方法受到的唯一限制是当任务集中存在“重型”任务时，参数化资源利用率界限需要低于一个上限。该方法的第二个好处是，由于其使用了响应时间分析而非第一组算法中的资源利用率界限来决定一个处理器上可以容纳的最大负载，因此获得了比第一组算法更好的平均性能。

（3）共享缓存敏感的多核调度算法的研究。本文提出了一种基于共享缓存隔离的多核实时调度算法。该方法使用软件缓存划分技术，通过与调度算法的结合来隔离同时执行的硬实时任务的缓存空间，从而避免他们的相互干扰。该算法的基本思想是利用操作系统中虚拟地址与物理地址的映射，来将共享缓存划分成若干个缓存块。一个任务可以使用指定个数的缓存块，因此我们可以独立地计算每个任务的 WCET。在运行时，一个任务只有在能够得到它需要的缓存块时才执行。同时，本文提出了如何对此类算法进行可调度性分析。本文首先提出一个基于线性规划的充分判定条件，然后通过放松线性规划的部分约束，获得一个时间复杂度为$O(N^2)$的充分判定条件。

1.4 本文的组织结构

本文共分 9 章，各章具体内容组织如下：

第 1 章为绪论，主要介绍了本文的研究背景及意义，包括实时嵌入式系统研究背景，多核实时调度主要内容和意义，分析了国内外研究现状与尚存在的问题，介绍了本文的主要内容，以及篇章结构。

第 2 章主要介绍实时调度算法的背景知识，介绍了实时调度中的任务模型，基本技术，以及多核实时调度中存在的基本问题，以及现有的理论成果。

第 3 章研究了可抢占的固定优先级全局调度的可调度性问题，通过建立一个与单处理机调度中的关键时刻类似的概念，来进行有效的可调度性分析，并通过实验来评价所提出方法的性能。

第 4 章研究了不可抢占的固定优先级全局调度的可调度性问题，为不可抢占调度建立一个新的“问题窗口”概念来进行可调度性分析，同时讨论了多核处理器调度中不可抢占调度可能优于可抢占调度的现象。

第 5 章研究了一种固定实例优先级的全局调度算法及相应的分析技术。该算法通过静态的有限计算可以为挖掘任务实例间的优先级关系，显著提高了系统的实时性能。

第 6 章研究准划分调度算法，提出了一种基于速率单调调度（RMS）的准划分调度算法将单处理机调度中著名的 Liu&Layland 资源利用率界限扩展到多处理机调度。

第 7 章使用准划分调度算法进一步将其他更高的参数化资源利用率界限扩展到多处理机调度，并使用响应时间分析来替代第 6 章算法中的资源利用率界限来决定一个处理器上可以容纳的最大负载，以获得更好的平均性能。

第 8 章研究了考虑共享缓存多核调度算法的研究，提出了一种基于共享缓存空间隔离的调度算法来避免并行执行的硬实时任务间的相互干扰，并给出了此调度算法的可调度性判定方法。

第 9 章总结全文，并提出了未来研究方向以及可以继续深入研究的内容。

第2章 多核实时调度研究背景

本章首先介绍了实时调度中广泛采用的任务模型，以及在后面的讨论中需要使用到的重要定义；之后从不同角度介绍了多处理机调度的分类以及评价一个调度算法优劣的各类方法；最后介绍了多处理机调度领域的研究现状与最新成果。

2.1 实时调度的任务模型与基本概念

嵌入式实时系统通常是由多个相互独立的任务（task）组成的。每个任务根据一定的间隔周期被循环地释放，其每一次释放都要在一定的时间内完成特定的工作。这样的任务可以表示成：

定义 2.1（周期性任务）：一个周期性任务（或简称任务）τ_i 是一个三元组 $\langle C_i, D_i, T_i \rangle$，其中各元素定义为：

- C_i：最坏情况执行时间（WCET），即任务每次其释放需要的最长执行时间。
- D_i：相对截止期。即 τ_i 每次释放后期望在 D_i 时间内完成执行。
- T_i：最短释放间隔（也称为周期）。τ_i 的两次释放之间需要至少 T_i 时间间隔。

如果一个任务 τ_i 满足 $D_i = T_i$，称 τ_i 为隐含截止期任务；如果 $D_i \le T_i$，称 τ_i 为限制截止期任务；如果 $D_i > T_i$，称 τ_i 为任意截止期任务。

根据一个任务 τ_i 的相 C_i 和 D_i 和，可以求得一个任务的松弛时间 $S_i = D_i - C_i$。

定义 2.2（资源利用率）：一个任务 τ_i 的资源利用率为：

$$U_i = \frac{C_i}{T_i}$$

资源利用率表示了一个任务所带来的负载。更准确的说，它表示了在一个很长的时间范围内执行该任务所需要处理器处理能力的最大比例。对于限制截止期任务或者任意截止期任务，还可以通过截止期而定义一个类似的概念：

定义 2.3（密度）：一个任务 τ_i 的密度为：

$$\delta_i = \max(\frac{C_i}{T_i}, \frac{C_i}{D_i})$$

下面介绍任务集。若干个相对独立的任务在一起组成了一个任务集：

定义 2.2（周期性任务集）：一个周期性任务集（或简称任务集）$\tau = \langle \tau_1, \tau_2, ..., \tau_N \rangle$ 由 N 个相互独立的周期性任务组成。

需要注意的是，本文研究的任务集中的各个任务是相互独立的。在实际系统中，各

个任务之间可能不是完全相互独立的，比如可能共享逻辑或硬件资源。与单处理调度相似，对于这样的系统，可以在调度算法之上增加资源共享协议来进行管理。本文将不考虑各个任务间的数据共享或依赖关系。

定义 2.4（资源利用率总和）：一个任务集τ的资源利用率为：

$$U(\tau)=\sum_{\tau_i\in\tau}\frac{C_i}{T_i}$$

在多处理机调度中，还可以通过考虑系统中处理器核心的数目对任务集的资源利用率总和进行正规化：

定义 2.5（正规化资源利用率总和）：一个任务集τ在 M 个处理器核心上的正规化资源利用率为：

$$U_M(\tau)=\frac{U(\tau)}{M}$$

如上文所述，一个任务的资源利用率表示了执行该任务所需处理能力的最大比例。容易看出，如果一个任务集的正规化资源利用率总和超过 1，则在运行时系统的负载会超过硬件平台所能提供的处理能力，因此会产生无限增长的未处理负载。因此，一个任务集的正规化资源利用率总和不超过 1 是其能正确执行的必要条件。

如果一个周期性任务集τ中所有的任务都是隐含截止期任务，则称τ为隐含截止期任务集；如果τ中所有的任务都是限制截止期任务，则称τ为限制截止期任务集；如果τ中含有至少一个任意截止期任务，则称τ为任意截止期任务集。

定义 2.6（任务实例）：每一个周期性任务会释放一系列（无穷多个）任务实例（Task Instance）。任务τ_i连续两个实例的释放间隔至少为T_i，每次个任务实例最多执行C_i时间。

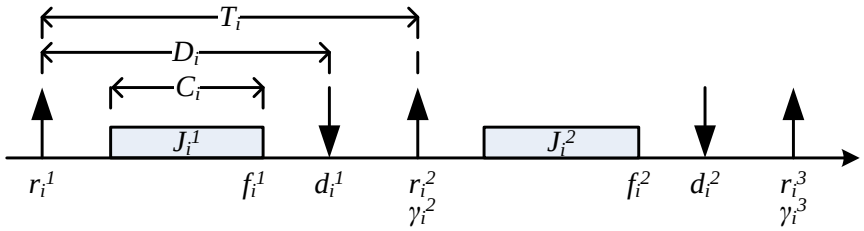
本文使用J_i^j来表示任务τ_i释放的第j个实例，或使用J_i来表示任务τ_i释放的某一个实例。实例J_i^j（J_i）的释放时间表示为r_i^j（r_i），绝对截止期表示为d_i^j（d_i）。一个实例必须在其绝对截止期之前完成执行，所以根据相对截止期D_i的定义可知$d_i^j=r_i^j+T_i$（$d_i^j=r_i^j+T_i$）。实例J_i^j（J_i）的完成时间，即该实例完成其执行的时刻，表示为f_i^j（f_i）。

定义 2.7（活跃实例）：一个实例J_i^j是活跃的，当且仅当其已经被释放但未完成执行。换言之，一个实例在时间区间$[r_i^j, f_i^j)$内是活跃的。

如果τ_i是一个限制截止期任务（包括时一个隐含截止期任务），则在任何时刻τ_i有至多有一个活跃实例。如果τ_i是一个任意截止期任务，则在某时刻τ_i可能有多个活跃实例。本文规定当一个任务有多个活跃实例时，其中较晚释放的实例必须等到所有较早释放的实例都完成执行以后才可以开始执行。这个限制被普遍地被现实中的实时操作系统

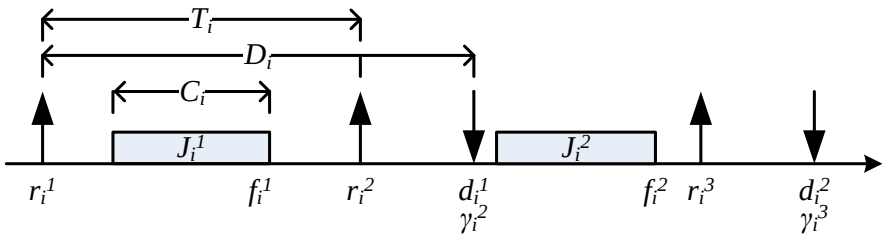
所采用，如 RTEMS[64]和 LITMUS^{RT}[65]。采用此限制有三方面的好处：一是简化操作系统的实现，二是节省内存空间，三是可以自动解决一个任务中各个实例直接的资源共享问题。根据此限制，本文为每个实例 J_i^j 进而定义一个就绪时间 $\gamma_i^j = \max(r_i^j, f_i^{j-1})$，其中 r_i^j 为 J_i^j 的释放时间，f_i^{j-1} 为 J_i^j 的前继实例 J_i^{j-1} 的完成时间。

定义 2.8（就绪实例）：一个实例 J_i^j 在时间区间 $[\gamma_i^j, f_i^j)$ 内是就绪的。



(a) 一个限制截止期任务

(a) A constrained-deadline task



(b) 一个任意截止期任务

(b) An arbitrary-deadline task

图 2.1 任务及任务实例相关概念示意图

Fig. 2.1 Illustration of related concepts of task and task instance

根据定义，可以知道任何任务在任意时刻至多有一个就绪实例。图 2.1 表示了以上定义的一些相关概念，其中图 2.1(a)为一个限制截止期任务，而图 2.1(b)为一个任意截止期任务。

2.2　多处理机调度算法分类

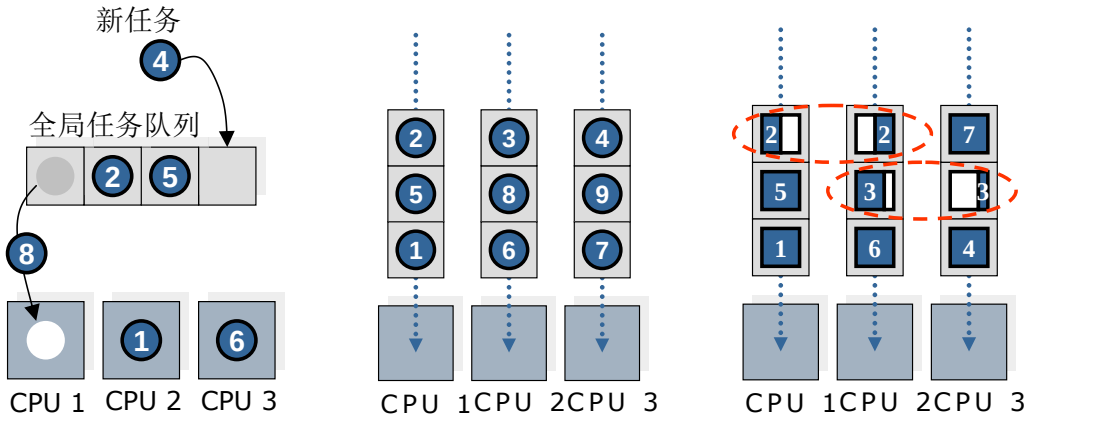
由于一个任务集中包含多个任务，操作系统需要决定在什么时间以及在哪个处理器上运行哪个任务，即进行任务调度。在过去的近 50 年时间里，有大量工作研究如何在多个处理器单元上进行任务调度。这些工作最开始是以多处理机系统（同一个主板上有若干个相互连接的处理器）为背景的，因此，这类调度问题一直被称为多处理机调度问题。虽然多核系统在许多方面与多处理机系统都有着本质的区别，但是从调度的最基本抽象层面，其依然继承了多处理机系统的本质特征，即需要在多个执行单元上分配任务的执行。因此，在针对多核处理器的实时调度研究领域，一般依然沿用多处理机调度来

作为这一类调度问题的统称，本文亦延用此名称。

在实时系统中，应用最为广泛的是基于优先级的调度。基于优先级调度的基本原则是，在所有就绪的任务实例中，选择优先级最高的那个来运行。本文的研究范围仅限于基于优先级的调度算法。在基于优先级调度的基本原则之上，可以对调度增加附加规则，因而形成许多不同种类的调度算法。下面从不同角度来对多处理机调度算法进行分类。

2.2.1 全局调度、划分调度与准划分调度

多处理机调度算法根据是否为每个处理机配置独立的调度器，可以分为两类：（1）全局调度（Global Scheduling）；（2）划分调度（Partitioned Scheduling）。在全局调度中，整个系统使用一个统一的任务队列，每个任务可以在任意一个处理器核心上执行，并可以在运行时从一个处理器核心迁移到另一个处理器上，如图 2.2(a)所示。在划分调度算法中，每个处理器核心拥有独立的任务队列。每个任务被分配到一个特定的处理器核心上，且只在该处理器上运行而不会迁移到其它处理器核心上。



(a) 全局调度
(a) Global scheduling

(b) 划分调度
(b) Partitioned scheduling

(c) 准划分调度
(c) Semi-partitioned scheduling

图 2.2 全局调度、划分调度与准划分调度示意图

Fig. 2.2 Illustration of global scheduling, partitioned scheduling and semi-partitioned scheduling

全局调度算法与划分调度算法各有优劣。首先，从操作系统实现的角度，全局调度算法的实现相对比较复杂，而划分调度算法基本上可以从单处理器操作系统上直接继承过来。现有的支持多核处理器的实时操作系统无一例外地都支持划分调度算法，而仅有一部分支持全局调度算法，例如 LITMUSRT[65]、QNX[66]、VxWorks[67]等。但是，随着多核处理器的更深入普及和多核操作系统的进一步发展，越来越多的操作系统有望支持全局调度算法。

从系统运行时开销的角度，全局调度由于需要共享调度器数据结构，因此调度器运

行的开销通常高于划分调度。但是实验表明，两种调度策略此项开销的差别并不是特别明显[68]。此外，全局调度由于需要在不同的处理器核心之间进行任务迁移，一般情况下其上下文切换的开销，尤其是缓存相关的开销要高于划分调度。现代多核处理器由于广泛使用片上高速共享缓存，两种调度策略此项开销的差别在多核处理器上对系统性能的影响也比较小[54, 68]。

从系统平均性能的角度，全局调度有更好的负载平衡特性。有待处理任务的情况下，全局调度器不会使处理器空闲而造成资源浪费，因此一般来讲对系统资源的利用率较高。但是在划分调度算法中，即使某个处理器核心上有待处理的任务，其他处理器核心依然可能空闲，从而造成系统资源浪费。

从调度算法分析的角度，全局调度的分析问题非常复杂，实际中设计者们为了保证安全性，只能使用悲观的近似分析，从而造成系统的过量设计（Over-Design）冗余。在划分调度中可以将各个处理器核心上所分配的任务看作一个单独的单处理机系统，而使用成熟的单处理机调度和分析技术。但是，由于划分调度算法不允许进行任务迁移，所以设计者需要进行仔细的任务划分以使各处理器间的负载尽可能平衡。任务的划分问题与背包问题类似[69]，是 NP 难问题。此外，由于受到与背包问题类似的限制，任何划分调度算法在最坏情况下都会浪费 50%的运算资源。如下例所示，考虑一个将一个含有 N 个任务的任务集，每个任务的资源利用率都为$0.5+\varepsilon$其中ε为一任意小的正数。容易看出，任务集中任意两个任务的资源利用率之和都大于 1，所以不能将两个任务分配到同一个处理器核心上。因此，需要 N 个处理器核心才可能使用划分调度算法正确调度这个任务集，而此时系统中所有任务的负载与系统所提供的运算能力之比为$(0.5+\varepsilon)/1$。

为了结合全局调度与划分调度各自的优点，研究者们提出了一种对划分调度的改进，称为准划分调度（Semi-Partitioned Scheduling）。准划分调度中，大部分的任务像在划分调度中一样，被预先分配到一个固定的处理器核心上运行，但是还有一小部分任务，可能通过某种受控的方式在多个处理器核心上运行，如图 2.2(c)所示。准划分调度最大的优势在于，它在既继承了划分调度优点的同时，增强了系统的负载平衡，而且可以突破划分调度最坏情况下浪费 50%运算资源的限制。

本文的研究内容中，第 3，4，5，8 章为全局调度，第 6，7 章为准划分调度。

2.2.2 固定任务优先级调度与固定实例优先级调度

根据一个任务的所有实例是使用统一的优先级还是有各自不同的优先级，调度算法可以分为两大类：（1）固定任务优先级调度（Task-Level Fixed-Priority Scheduling）；（2）固定实例优先级调度（Job-Level Fixed-Priority Scheduling）。

在固定任务优先级调度中，每个任务被赋予一个优先级。一个任务释放的所有实例都使用该优先级参与调度。例如，著名的单调速率调度算法（RMS）就是一种固定任务优先级调度策略，其根据任务的周期长短来为任务分配优先级：周期越短，优先级越高。

在固定实例优先级调度中，每个任务实例被赋予一个优先级。例如，著名的最早截止期优先（EDF）就是一种固定实例优先级调度策略，其根据任务实例绝对截止期的时间先后顺序来为任务实例分配优先级：绝对截止期越早，优先级越高。

本文的研究内容中，除了第 5 章以外，都为固定任务优先级调度。从现在开始，将简称固定任务优先级调度为固定优先级调度（Fixed-Priority Scheduling）。为了简化叙述，本文假设固定优先级调度中每个任务具有一个独享的优先级，但是本文的所有结果都可以很容易地扩展到多个任务共享一个优先级的情况。本文使用任务的下标来表示任务的优先级顺序，且根据惯例，一个较小的任务下标值代表一个较高的优先级，即：

$$i < j \Leftrightarrow \tau_i \text{的优先级高于} \tau_j \text{的优先级}$$

第 5 章研究固定实例优先级调度。使用 $\mathrm{prt}(J)$ 来表示一个任务实例 J 的优先级。使用 $p \rhd q$ 表示优先级 p 高于优先级 q。类似地使用 $p \lhd q$ 表示优先级 p 低于优先级 q，使 $p \lhd q$ 表示优先级 p 低于优先级 q 用。此外，使用 $\perp$ 表示一个比所有由自然数表示的优先级都低的最低优先级。

2.2.3 可抢占调度与不可抢占调度

根据一个（高优先级）任务实例是否可以强迫其它（低优先级）任务实例交出处理器的使用权，调度算法可以分为两大类：（1）可抢占调度（Preemptive Scheduling）；（2）不可抢占调度（Non-Preemptive Scheduling）。例如，一个低优先级任务实例在时刻 0 释放且需要执行 4 个时间单位，而一个高优先级任务实例在时刻 2 释放。在可抢占调度中，在时刻 2，高优先级任务实例将抢占（Preempt）低优先级任务实例，即低优先级任务实例将被挂起（Suspended）处理器转而执行高优先级任务实例，待到高优先级任务实例执行结束，低优先级任务实例将恢复执行（Resume）。而在不可抢占调度中，低优先级任务实例将执行至结束（至时刻 4），之后高优先级任务实例才能开始执行。

一般来讲，可抢占调度比不可抢占调度具有更好的响应性能，因为一个更重要或更紧急的任务实例可以马上得以执行而不必等待低优先级任务实例执行结束。但是，不可抢占调度也具有许多优点：不可抢占调度比抢占调度更容易实现，且运行时的开销更低；在抢占调度中，由于缓存和流水线等硬件结构，对抢占造成的开销往往非常难以估计。不可抢占调度的优势在多核处理器平台上变得更加明显：在多核处理器上，由于任务迁移所引起的开销更高，且更加难以预测。但是，这些问题在不可抢占调度策略中则容易

处理得多，因为每个任务实例都会一直执行到结束而任务迁移只发生在任务实例边界上。本文的研究内容中，第 3，6，7 章为可抢占调度，第 4，8 章为不可抢占调度，第 5 章为两种策略的混合方式。

2.2.4 主动空闲调度与非主动空闲调度

根据是否允许在有等待执行的任务时使处理器空闲，调度算法可以分为：（1）非主动空闲调度（Work-Conserving Scheduling）；（2）主动空闲调度（Non-Work-Conserving Scheduling）。本文的研究内容中，所有基于全局调度的算法（第 3，4，5，8 章）都为非主动空闲调度。

对于划分调度或准划分调度，广义上来讲它们属于主动空闲调度，这是因为从整个系统的角度看，及某些处理核心上有正在等待执行的任务，而其它处理核心却处于空闲状态。但是如果只针对其中的某一个处理器核心来看，划分调度或准划分调度可以是局部非主动空闲的，即一个处理器核心在存在本地待执行任务时不允许空闲。本文的研究内容中，所有基于准划分调度的算法（第 6，7 章）都为局部非主动空闲调度。

2.3 可调度性判定

实时调度的目的是保证系统的时间正确性，即使每个任务实例都能在其绝对截止期之前完成，称为满足截止期；否则称为错失截止期。对于硬实时系统，需要在系统开始运行之前，即系统设计阶段就能绝对保障系统的时间正确性，这就需要根据系统中任务的参数和所采用调度算法的规则来分析系统的可调度性。定义如下：

定义 2.9（可调度性）：使用某一个调度算法 A 来调度一个任务集 τ。一个实例可以被 A 调度的当且仅当该实例在其绝对截止期前完成执行。一个任务是可被 A 调度的当且仅当其释放的（无穷多个）实例均为可被 A 调度的。一个任务集 τ 是可被 A 调度的当且仅当其所有任务均为可被 A 调度的。

2.3.1 响应时间分析与资源利用率界限

判定系统可调度性的方法主要有以下两类：（1）基于响应时间分析的判定；（2）基于资源利用率界限的判定。

（1）基于响应时间分析的可调度性判定

定义 2.10（响应时间）：一个任务实例 J_i^j 的响应时间（Response Time），记为 R_i^j，是其释放时间 r_i^j 和执行结束时间 f_i^j 之间的时间距离，即

$$R_i^j = f_i^j - r_i^j$$

一个任务τ_i的最坏响应时间（Worst-Case Response Time，WCRT），记为R_i，是其所释放的所有实例的响应时间的最大值。简称任务的最坏响应时间为其响应时间。

通过计算一个任务的响应时间并与其相对截止期相比较便可判断该任务是否可调度：满足$R_i \le D_i$则任务可调度，否则任务不可调度。通过将此过程应用于任务集中的每一个任务，便可以判断整个系统的可调度性。这种方法称为基于响应时间分析的判定。

对于许多调度算法，无法（高效地）计算每个任务响应时间的准确值，这时可以通过求得一个任务响应时间的上限值来进行充分而非必要的可调度性判定：如果这个上限值小于其相对截止期，则该任务在运行时一定可被调度。充分而非必要的可调度性判定也称为安全的可调度性判定。当一个任务被一个充分而非必要的可调度性判定认定为不可调度时，其在运行时可能实际为可调度。基于响应时间分析的判定中一种常用的方法为直接假设一个任务的响应时间等于其相对截止期，然后来判断该假设是否会导致矛盾。如果不能，则可知该任务的截止期为其响应时间的安全上限，因此它是可调度的。

（2）基于资源利用率界限的可调度性判定

定义 2.11（资源利用率界限）：一个调度算法A的资源利用率界限为一个正数UB，使得任意满足下述条件的任务集τ都是可调度的：

$$U_M(\tau) \le UB$$

单处理机调度中有两个著名的资源利用率界限[14]：（1）RMS 的资源利用率界限为 69.3%；（2）EDF 的资源利用率界限为 100%。资源利用率界限提供了一种非常高效的可调度性判定：判定过程只需要简单对所有任务的资源利用率进行求和并和UB进行比较。这种判定方法非常适合于系统运行时的准入控制（Admission Control），即在系统正在运行时决定一个新到来的任务是否可以被加入系统而不引起任何任务错失截止期。此外，这种方法还非常适合于进行系统设计空间探索（Design Space Exploration），即寻找一组不但使系统可调度还针对一定目标进行优化的系统参数。除了用于系统的可调度性判定，资源利用率界限还是一个可以直接用来评价调度算法质量的标准：一般来讲，资源利用率界限越高的调度算法越好，因为它可以使更多的任务集可调度。

基于资源利用率界限的可调度性判定还可以通过考虑所判定任务集参数特征来提高判定的准确性。比如，RMS 的资源利用率界限可以通过考虑任务集中任务的个数N而成为$N \times (2^{1/N} - 1)$。例如，当$N = 2$时该界限大约为 82.8%，其高于 RMS 的一般性资源利用率界限 69.3%，因此对含有两个任务的任务集进行的判定更加准确。这种资源利用率界限称为参数化资源利用率界限。在第 7 章中将详细介绍参数化资源利用率界限的定义和相关性质。

2.3.2 可调度性判定的可持续性

Baruah 和 Burns 介绍了可持续性（Sustaintability）的概念[70]，这个概念也早期的文献中也称为鲁棒性（Robustness）或可预测性（Predictability）。

定义 2.12（可持续性）：对于一个可调度性判定条件，如果任何被该可调度性判定条件认定为可调度的任务集，将其中某些任务的执行时间缩短（或周期变长，或相对截止期变长）以后，该任务集依然是可调度的，则称这个可调度性判定条件是关于执行时间（或周期，或相对截止期）可持续的。

直观上讲，如果一个可持续的可调度性判定条件认定一个任务集为可调度的，那么如果系统在运行时的行为“好于”其最坏可能性，则该任务集一定是可调度的。可持续性是一个很重要的性质，因为它使得系统设计者在分析系统的可调度性时只需考虑系统的最坏情况参数，而不必系统所有可能表现出来的参数值，这大大地降低了系统分析的复杂度。

Baker 和 Baruah 又介绍了一个更强的概念——自可持续性[71]：

定义 2.13（自可持续性）：对于一个可调度性判定条件，如果任何被该可调度性判定条件认定为可调度的任务集，将其中某些任务的执行时间缩短（或周期变长，或相对截止期变长）以后，该任务集依然可以被该可调度性判定条件是判定为是可调度的，则这个可调度性判定条件为关于执行时间（或周期，或相对截止期）为可持续的。

容易看出，一个自可持续的可调度性判定条件一定是可持续的，因此自可持续性是一个比可持续性更强的性质。自可持续性在增量设计过程中有重要作用。如果一个可调度性判定条件是自可持续的，则在设计过程中如果将一个被认定为可调度的任务集的参数变得更好（执行时间缩短，或周期变长，或相对截止期变长），则该任务集依然是可以被验证为可调度的。因此具有这类性质的可调度性判定条件更适合于应用于增量的系统设计空间探索。本文中所研究的调度算法，若非特殊说明都为自可持续的。

2.3.3 调度算法和可调度性判定的质量评价

如 2.2 节所述，设计多处理机调度算法有许多种不同的选择，对于同一种调度算法，可能存在多种不同的可调度性判定条件。本节将介绍几种常见的可调度性判定质量评价方法用于比较不同方法的优劣。

首先，要明确调度算法的质量和可调度性判定条件之间的关系。从广义上讲，一个调度算法的质量（又称实时性能）是指客观上该调度算法能成功调度实时任务集的能力。比如，可以知道（本文假设的任务模型下）单处理机调度中 EDF 的质量高于 RMS 的质

量，因为任何可以被 RMS 调度的任务集都可以被 EDF 调度。但是，在很多情况下，并不能够确切知道一个调度算法 A 是否可以成功调度某个任务集，即不知道调度算法 A 的充分必要可调度性判定条件，这时，就无法对算法 A 客观上成功调度实时任务集的能力进行评估。对于这种情况，只能通过已知的算法 A 的充分可调度性判定条件的质量来间接评价算法 A 的质量。例如，有两个调度算法 A 和 B，客观上 A 的质量优于 B，但是对于 A 只能得到一个质量很低的充分判定条件，而为 B 建立的充分判定条件的质量较高且优于 A，此时，对于系统设计者而言，为了使用较少的硬件资源而保证系统的可调度性，B 将是优于 A 的选择。换言之，对于系统设计者而言，一个调度算法的质量取决于其能被某个已知可调度性判定条件保证的质量，称为可分析质量（又称可分析性能）。除非特殊说明，本文中所指的调度算法的质量为其已知的可调度性判定条件的质量。因此，调度算法质量的评价也统一成对可调度性判定条件质量的评价。

有些情况下，两个可调度性判定条件之间有清晰的优劣关系。比如，如果任何一个被可调度性判定条件 A 接受的任务集（即认定为可调度的任务集）都能被判定条件 B 接受，那么称判定条件 B 优于性判定条件 A。若在此基础上，存在某些任务集可以被判定条件 B 接受而不能被 A 接受，则称判定条件 B 严格优于判定条件 A。

但是在很多情况下，两个可调度性判定条件不存在清晰的优劣关系。比如有两个可调度性判定条件 A 和 B，既存在 A 可以接受而 B 不能接受的任务集，也存在 B 可以接受而 A 不能接受的任务集，此时称 A 和 B 为不可直接比较的。我们称两个调度算法为不可直接比较的，若他们的可调度性判定条件是不可直接比较的。对于这种情况，下面几个指标为常用的调度算法或可调度性判定质量评价依据。

（1）资源利用率界限

如第 2.3 节介绍，资源利用率界限可以被直接用作一种高效的可调度性判定条件。因此，可以通过比较两个调度算法已知的最好资源利用率界限来对决定它们的质量优劣。资源利用率界限越高的调度算法质量越高。

（2）加速因子

这种方法是通过某调度算法与（客观存在的）最优算法直接的比较关系，来间接比较不同的调度算法。加速因子的定义如下：

定义 2.14（加速因子）：一个调度算法 A 的加速因子为 s，如果其满足如下条件：任何在处理器速度为 1 的硬件平台上可被（客观存在的）最优调度算法调度的任务集在速度为 s 的硬件平台上都可以被算法 A 调度。

直观上讲，加速因子表明需要用一个多快的处理器才能补偿某个算法的次优性。加速因子越小，调度算法的质量越高。当 $s=1$ 时，调度算法即为最优。如果已知一个调度

算法的资源利用率界限UB，则该算法存在一个加速因子$1/UB$（根据最优算法的资源利用率界限不能超过 1 这一事实可证）但是，一个调度算法的加速因子一般情况下不能转化为资源利用率界限。可见，加速因子是一个比资源利用率界限更弱的性质。

（3）接受率

上述两个指标都是基于调度算法（的可调度性判定条件）的最坏情况所建立的性质。例如虽然单处理机 RMS 的资源利用率界限为 69.3%，但大量实践表明大部分资源利用率总和在 80%左右的任务集是可以被 RMS 调度的。再比如，由于 Dhall 效应，多处理机全局 EDF 的资源利用率界限为 0，但这并不意味着多处理机全局 EDF 不能调度任何任务集，相反，有很多具有较高资源利用率的任务集是可以被多处理机全局 EDF 调度的。这是因为用于构建调度算法资源利用率界限的情况可能是非常特殊，甚至是病态的（Pathological），这些极端情况在实际系统设计中几乎不会出现，因此仅仅使用这类最坏情况下的性质可能无法对调度算法和可调度性判定在实际系统中所体现出来的质量进行准确评价。

针对这个问题，可以通过使用可调度性判定对大量的任务集例子来进行判断，来观察其能够将其中多少比例的任务集认定为可调度的，称为接受率。一般来讲，接受率越高，可调度性判定的质量越高。由于这是一种基于统计的评价方法，其结果严重依赖于对样本空间的选择。例如，如果所有参与测试的任务集的正规化资源利用率总和都超过 1，那么任何可调度性判定都具有相同的接受率，即为 0，而不提供任何有用的信息。因此，为了进行相对客观准确的评价，样本空间的设定是非常重要的。

本文将采用文献[72]中的方法来构建任务集的样本。该方法总的原则是：（1）通过设定每个任务的各个参数范围来随机生成测试任务集；（2）对具有不同正规化资源利用率总和的任务集的接受率进行分别考察。

其任务生成过程具体如下：对于每个任务，分别设定其最坏情况执行时间，周期，资源利用率，相对截止期与周期的比例等参数的范围，然后在相应的范围内随机选取参数，来生成一个任务。其任务集生成的过程如下：首先生成一个含有$M+1$个任务的任务集来进行测试，然后每次生成一个比上一次任务数多 1 个的任务集进行测试。重复这个过程，直到所生成任务的正规化资源利用率总和超过 1，此时将任务个数重置为$M+1$并重复上面整个过程，直到生成了一个足够大的样本空间。这种任务集生成方法的好处是可以使生成的任务集比较平均地分布在不同的（正规化）资源利用率总和区间。

对所生成任务集测试结果的表达方式如下：将总的（正规化）资源利用率总和区间分为若干个大小相等的段，对每一个区间段内任务集，计算其中被认定为可调度的任务所占比例，即为该（正规化）资源利用率总和区间的接受率。

以上介绍了三种常用的调度算法或可调度性判定质量评价指标。这三种指标各自有优势和不足。本文将主要采用资源利用率界限和接受率这两个指标对所研究调度算法和可调度性判定条件进行质量评估。

2.4 现有理论成果概述

2.4.1 全局调度

Liu 于 1969 年指出对全局多处理机调度分析的难度远高于对单处理机调度的分析：即使多个处理器空闲时一个任务也只能在其中一个上面执行，这一简单的实时为在多个处理器上调度进行调度增加了惊人的难度[16]。如果将单处理机调度中的最优调度算法 RMS 或者 EDF 应用于全局多处理机调度中，则会产生一种称为 Dhall 效应的现象，即存在正规化资源利用率总和任意接近于 0 的任务集在全局 RMS 或 EDF 下是不可调度的。换言之，全局 RMS 和 EDF 的资源利用率界限无限接近于 0。这一结论，对研究领域对全局调度的的观点产生了深远的影响。在很长一段时期内，学术界普遍认为全局调度是不适合实时系统的。

克服 Dhall 效应的一种方法是使用基于公平的调度（Fairness Scheduling）[73]。这种方法把任务的执行切成小块，让各个任务频繁交替执行并使每个任务执行的进度与其负载成正比。基于公平的调度及其各种改进版本[74~85]，理论上能够达到可调度性的最优，但通常被认为是不适合实际应用的。这主要归咎于其所引起的大量的上下文切换进而造成巨大的运行时开销。

对于全局调度的负面观点自从 Phillips 等人于 1997 年发表的论文[26]后得到了很大改变。这篇论文表明，Dhall 效应是由资源利用率非常高的任务造成的，而且可以通过增加处理器的速度（而不是增加处理器的个数）来得到补偿。这一思想随后对全局调度算法的设计和分析进行带来了重大的进展，其中包括一些针对仅含有资源利用率较小的任务集的全局调度算法的分析技术，以及通过赋予高资源利用率任务较高优先级的来设计新的全局调度算法以提高资源利用率界限或者降低加速因子[27, 29, 86~89]。这些工作，从理论层面显示了全局调度在多核实时系统中的应用潜力。

对于全局调度进行精确分析从而充分发掘其潜力的主要障碍是其所表现出的许多不规则性（Anomaly），即将一个原本可调度的任务集的参数变得“更好”反而可能使其不可调度。一个著名的例子是，全局调度中的关键时刻（Critical Instant）是未知的。广义上讲，关键时刻是一个引起某任务具有最坏相应时间的具体情形。比如在单处理机固定优先级调度中，关键时刻为所有任务同时释放第一实例，且每个任务的都尽可能快地

释放。通过考查这个关键时刻，便可以建立对单处理机固定优先级调度的精确分析。虽然从直观上，单处理机固定优先级调度中的这个关键时刻似乎总是会引起系统的最大负载，但是其对全局调度并不成立[90,91]。因此，对于全局调度的精确分析原则上需要对系统所有可能的行为进行考查。一些工作使用显式或隐式的状态空间枚举来对全局调度的可调度性进行分析[92~94]，但是由于其运算复杂度非常高，在可承受的时间内仅能处理很小规模的任务系统（每个任务集不超过 10 个任务，且对任务的参数有严格限制）。因此，使用基于全状态空间枚举的精确分析方法无法应用于实际系统。

研究者们在安全近似的全局调度分析方面做了大量的工作。这些工作的共同方法是计算单个任务在某个时间区域内工作量的上限，然后对各个任务进行求和得到整个系统在该时间区域内工作量总和的上限，并以此进行可调度性分析。其中大部分工作致力于如何通过排除不可能的系统行为来使上述计算变得更加精确[28, 29, 34, 39~43]。

单处理机调度中最优的调度算法 RMS 和 EDF 在多处理机全局调度中不但丧失了最优性，而且可能导致非常差的实时性能。因此产生了一个根本性的问题：什么样算法才是好的全局调度算法？Anderson 等人的研究表明，尽管全局 EDF 不能在系统满负载的情况下（正规化资源利用率为 100%）保证所有任务的截止期，其依旧维持着某种意义上的一种“最优性”：在满负载情况下全局 EDF 保证每个任务的响应时间都是有限的。与此相反，在满负载情况下全局固定优先级调度中任务的响应时间可能是无限的[95]。

研究者们还提出了一些设计原则和附加机制来提高全局固定优先级和 EDF 调度算法的实时性能。例如，在固定优先级调度中选择合适的优先级顺序[96,97]，在运行时提升那些马上就要错失截止期的任务实例的优先级[98~101]以及强迫将一部分负载放到一定存在空闲处理器的资源的时间区域执行[102]等。

2.4.2 划分和准划分调度

对任务集在多个处理器上进行最优的划分是 NP 难问题。研究者们提出了许多启发式划分算法来获得次优解。Andersson 等人在[103]中证明对于隐式截止期周期任务集任何划分调度算法的资源利用率界限最坏情况都不超过 $(m+1)/2$。 Burchard 等人提出了一种基于 RMS 的划分调度算法 RMST[104]，并证明了其资源利用率界限为：

$$(m-2)(1-U_{\max})+1-\ln 2$$

其中 $U_{\max}$ 是所有任务的最大资源利用率。然后，他们又提出了 RMST 的扩展版 RMGT，其通过根据每个任务的资源利用率是否超过 1/3 来把所有任务分为两组进行划分，并证明 RMGT 的资源利用率界限为：

$$\frac{1}{2}(m-\frac{5}{2}\ln 2+\frac{1}{3})\approx 0.5(m-1.42)$$

Oh 和 Baker 证明了划分固定优先级调度算法的资源利用率界限不可能超过上限[105] $(m+1)/(1+2^{1/(m+1)})$。Lopez 等人随后对上述结果进行了推广[106,107]，提出了 RMFFDU，RMBF，RMFF，RMWF 等划分固定优先级算法，并为它们各自证明了关于任务个数和最大任务资源利用率$U_{\max}$的资源利用率界限。最终，Andersson 和 Jonsson 在 2003 年设计了一个划分固定优先级调度算法 RBOUND-MP-NFR[108]，并证明其资源利用率界限为 M/2，也就是任意划分调度算法资源利用率界限的上限值。

在基于 EDF 的划分调度方面，Lopze 等人证明了任何不故意浪费处理器资源的划分算法结合 EDF 调度的资源利用率界限的下限为

$$m-(m-1)U_{\max}$$

且上限为

$$\frac{\lfloor 1/U_{\max}\rfloor M+1}{\lfloor 1/U_{\max}\rfloor+1}$$

其中假设$N>M/(\lfloor 1/U_{\max}\rfloor)$。他们还证明了所有根据资源利用率递减顺序进行分配的划分 EDF 算法都能到达这个上限值，例如 EDF-BF 和 EDF-FF。

Anderson 等人首次提出准划分调度算法[109,110]，该算法基于 EDF 调度且主要针对软实时系统。第一个针对硬实时系统的准划分调度算法是 EKG[44]。该算法通过将某些任务划分成若干部分并让每一个部分在特定的时间窗口里执行来解决被切分任务的同步问题。该算法的资源利用率界限取决于任务进行切换的频繁程度，比如当每个任务实例平均被抢占 4 次时，其资源利用率界限为 66%。随后 EKG 被扩展到任意截止期等任务模型，其共同特点都是通过更频繁的任务切换来获得较高的资源利用率界限。

Kato 等人提出了一系列基于优先级的准划分调度算法。基于优先级的准划分调度算法与 EKG 类算法相比，具有较小的运行时开销，因此更适合实际应用。其中，Kato 等人提出的基于 EDF 的算法 EDDF[49]的资源利用率界限可达 65%。在基于固定优先级的算法方面，Kato 等人提出了 RMDP 和 DMPM[50, 52]。这个两算法的资源利用率界限都为 50%，即这些算法在最坏情况下与划分调度算法的资源利用率界限相同。

Lakshmanan 等人提出了一个基于 DMS（单独截止期调度）的准划分调度算法 PDMS_HPTS_DS[54]，该算法的资源利用率界限为 65%，且当任务集中所有任务的资源利用率都比较小时，可以达到 69.3%。

第3章 可抢占全局固定优先级调度分析

本章采用基于响应时间分析的方法研究可抢占全局固定优先级调度算法的可调度性判定问题。单处理机调度中的响应时间分析技术在过去 20 年间已经发展得非常成熟，并已经被扩展到许多更加复杂的任务系统模型[111~115]，并在实际中得到了广泛应用。相比之下，对于全局多处理机调度的响应时间分析方面的工作还很少。本章提出的响应时间分析方法是基于文献[34, 41]中的分析技术。具体地说，是通过将[34]中的问题窗口扩展应用于[41]中的响应时间分析方法，来得到一个更精确的，即适用于限制截止期，也适用于任意截止期任务集的响应时间分析方法。需要注意的是，本章提出的响应时间分析方法并不是[34, 41]中技术的简单结合。本章提出的响应时间分析方法实际上为全局固定优先级建立了一个与关键时刻相似的概念，称为近似关键时刻。通过建立近似关键时刻，可以不需要像[34]那样枚举所有可能的问题窗口大小就能更加准确地估计任务的响应时间上限。与现有工作相比，本章提出的响应时间分析方法的创新点主要体现在：（1）本章提出的方法不但在理论上严格优于现有方法，且通过实验表明其具有明显的实时性能优势；（2）本章提出的方法是第一个能够处理任意截止期任务集的响应时间分析方法，且证明了全局固定优先级调度中任务具有有限响应时间的一般性条件。

3.1　响应时间分析技术概述

3.1.1 单处理机上限制截止期任务集

经典的单处理机系统响应时间分析技术是在[116]中首次提出的。其分析方法只适用于限制截止期任务集（即 $\forall \tau_i \in \tau : D_i \le T_i$）。

对于一个任务 τ_k 进行响应时间分析要基于 $k-$忙碌期概念。直观上讲，“ $k-$忙碌期”是一个满足如下条件的最大连续时间区域：在该时间区域内的任意时间点处理器都在执行 τ_k 或比 τ_k 优先级高的任务。在单处理机固定优先级调度中，关键时刻定义了导致最大响应时间的任务释放时间点：τ_k 与所有比 τ_k 优先级高的任务同时释放的时间点。这一时间点正是 $k-$忙碌期的起始点。因此，τ_k 在长度为 x 的 $k-$忙碌期内所承受的干涉为 $\sum_{i<k}\lceil x/T_i \rceil \cdot C_i$。可以通过寻找下述迭代式的最小非负解来计算 τ_k 的最坏响应时间：

$$x = \sum_{i<k} \left\lceil \frac{x}{T_i} \right\rceil \cdot C_i + C_k$$

由于上式的等式右侧为关于 x 的单调递增函数，又知 τ_k 的响应时间上限一定不小于

C_k，因此上式的最小非负解可以通过从$x = C_k$开始迭代寻找其最小不动点进行求解。

3.1.2 多处理机上的限制截止期任务集

上一小节中响应时间分析的思想也可以应用于多处理机系统。单处理机系统与多处理机系统的主要区别在于，在多处理机调度中的的关键时刻是未知的，因此无法精确地计算τ_k在k–忙碌期内所承受的干涉，而只能近似地获得一个上限。

一个高优先级任务τ_i在k–忙碌期内的最大工作量可以被分为三部分：

前部任务实例工作量（carry-in）：所有在k–忙碌期开始前释放，且绝对截止期在k–忙碌期之内的任务实例（前部任务实例）所贡献的工作量。最多存在一个前部任务实例。

中部任务实例工作量（body）：所有释放时间与绝对截止期都在k–忙碌期内的任务实例（中部任务实例）所贡献的工作量。

后部任务实例工作量（carry-out）：所有在k–忙碌期结束前释放，且绝对截止期在k–忙碌期结束之后的任务实例（后部任务实例）所贡献的工作量。最多存在一个后部任务实例。

获得每个任务τ_i在一个长度为x的k–忙碌期内最大工作量的简单方式是假设前部工作量与后部工作量都为C_i：

$$W_k^{naive}(\tau_i, x) = \left\lceil \frac{x}{T_i} \right\rceil C_i + C_i$$

将所有高优先级任务的上述工作量上限相加，便可以获得所有高优先级任务工作量之和的一个上限。因此，可以使用$\frac{1}{M}\sum_{i<k} W_k^{naive}(\tau_i, x)$作为$\tau_k$在长度为$x$的$k$–忙碌期因高优先级工作量所承受的干涉时间。因此，可以通过计算下述等式的最小解来获得τ_k的响应时间上限[117~119]：

$$x = \frac{1}{M}\sum_{i<k}\left(\left\lceil \frac{x}{T_i} \right\rceil C_i + C_i\right) + C_k$$

Bertogna 等人将上述方法进行了重要改进[41]。本文将他们改进后的响应时间方法称为[BC-RTA]。[BC-RTA]与上述响应时间分析方法的主要区别如下。首先，在[BC-RTA]中，计算每个高优先级任务τ_i的工作量上限时，不是简单地假设其前部任务实例工作量与后部任务实例工作量都为C_i，而是对其导致最大工作量的情况进行仔细分析，从而获得更加精确的上限值。其次，文献[41]中观察到当τ_i在k–忙碌期内的工作量“过大”时，并非其所有的工作量都会引起对τ_k的干涉，这是因为τ_i的工作量超出的部分一定会和τ_k并行执行。这其实是单处理机调度和多处理机调度的一个本质区别。更详细地讲，文献

[41]中定义了一个高优先级任务 τ_i 在长度为 x 的 $k-$忙碌期内对 τ_k 产生的干涉时间：

$$I_k(\tau_i,x)=W_k(\tau_i,x)\Big|_0^{x-C_k+1} \tag{3.1}$$

其中使用 $A\ {}_B=\max(A,B)$，$A^{\ C}=\min(A,C)$，$A\ {}_B^C=\left(A\ {}_B\right)^C=A^{\ C}\ {}_B$。

用任务的干涉时间替代其工作量，可以将上述的响应时间分析方法变成：

$$x=\left\lfloor\frac{1}{M}\sum_{i<k}I_k(\tau_i,x)\right\rfloor+C_k \tag{3.2}$$

需要注意的是，在(3.1)中 τ_i 的干涉时间的上限是 $(x-C_k+1)$，而不是 $(x-C_k)$。这是因为如果使用 $(x-C_k)$ 作为干涉时间上限，则(3.2)的最小解不能保证是 τ_k 的响应时间上限。直观上讲，“+1”是用来保证当更多干涉时间可能阻止 τ_k 的执行时，等式(3.2)的右边会保持增长。比如，如果使用 $(x-C_k)$ 作为上限，当对(3.2)解的迭代搜索从 $x=C_k$ 开始时，对于每个高优先级任务 $\min\left(W_k(\tau_i,x),x-C_k\right)$ 一定为 0，因此搜索过程会立即结束。文献[41]中对这个问题进行了更详细的讨论和形式化的证明。

3.1.3 单处理机上的任意截止期任务集

单处理机调度的响应时间分析还被扩展到了任意截止期任务集（即允许 $D_i>T_i$ 的情况）[112]。对于任意截止期任务集，$k-$忙碌期的概念通过如下方式进行了扩展：当 τ_k 的一个任务实例在释放时，之前释放的所有 τ_k 的任务实例都已经完成，则一个新的 $k-$忙碌期开始。不失一般性，将 τ_k 引起一个新 $k-$忙碌期的任务实例记为 J_k^1，将其释放时间记为 r_k^1。因为在 r_k^1 时刻，τ_k 之前释放的所有任务实例都已经完成，J_k^1 在 r_k^1 时刻立即就绪（即 $\gamma_k^1=r_k^1$）。$k-$忙碌期将持续到 τ_k 接下来第一个任务实例在其下次释放时间前执行结束，并将这个使 $k-$忙碌期结束的任务实例记为 J_k^H，如图 3.1 所示。需要注意的是，$k-$忙碌期可能跨越 τ_k 的多个周期。

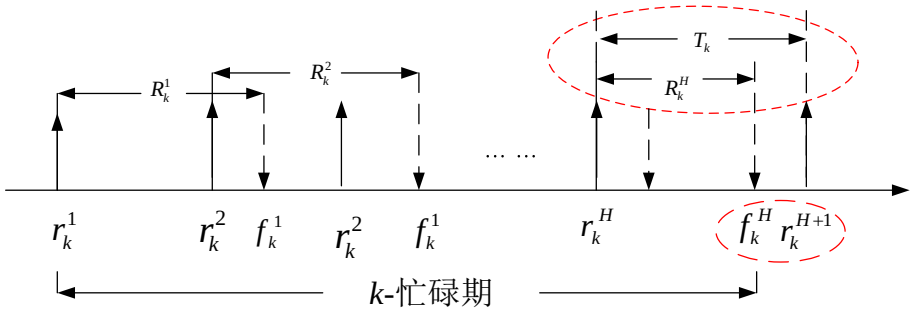


图 3.1　任意截止期任务在单处理机上 $k-$忙碌期示意图

Fig. 3.1 Illustration of level-k busy period for singleprocessor scheduling with arbitrary deadlines

为了计算每个任务实例 J_k^h（$h \in \{1,...,H\}$）的响应时间，将从 h = 1 开始解下述等式

$$x = \sum_{i<k} \left\lceil \frac{x}{T_i} \right\rceil \cdot C_i + h \cdot C_k \tag{3.3}$$

每次获得一个解之后，便使 h 增加 1。用 χ^h 来表示上述过程中第 h 步获得的解，则任务实例 J_k^h 的响应时间上限为：

$$R_k^h = \chi^h - (h-1) \cdot T_i$$

整个计算过程通过不断增加 h 循环进行，直到第一次得到满足如下终止条件 $Term(h,k)$ 的 χ^h：

$$Term(h,k): \chi^h \le h \cdot T_k \tag{3.4}$$

相应地可知 $H = \min\{h \ge 1 | Term(h,k)\}$。

τ_k 的最坏响应时间为 τ_k 在一个 k-忙碌期内所有任务实例响应时间的最大值。因此，τ_k 的响应时间上限可以通过下式求得：

$$R_k = \max_{h \in \{1,...,H\}} \{R_k^h\}$$

上述计算过程存在一个重要问题：对于（3.4）是否存在一个满足终止条件 $Term(H,k)$ 的 H，即整个计算过程是否能够终止。事实上可以很容易地证明当系统满足如下条件时一定存在一个满足条件 $Term(H,k)$ 的 H，即整个计算过程一定终止。

3.1.4 本章的主要创新点

第 3.1.2 节中介绍的目前最先进的多处理机调度响应时间分析方法[BC-RTA]比其它的可调度性分析方法具有更高的精确度，但是它仍然存在以下两方面的不足：（1）其分析结果依旧比较悲观；（2）其无法处理任意截止期任务集。本章针对这两方面不足提出了一种新的响应时间分析方法。与[BC-RTA]相比，该方法主要有以下两方面创新点：

（1）在第 4 节中，通过使用问题窗口扩展技术[34]，在不降低分析效率的前提下，增强对干涉时间估计的准确性，从而获得更精确的响应时间上限。该新响应时间分析技术在理论上严格优于[BC-RTA]（任何可以被[BC-RTA]判定为可调度的任务集一定可以该新分析方法判断为可调度）。此外，实验表明：平均情况下新分析方法的可调度性判定精确度大大优于[BC-RTA]（以及其它所有针对此问题的分析方法）。

（2）在第 5 节中将上述新分析技术扩展到可以处理任意截止期任务集的的情况。其中响应时间分析过程的终止性问题，即建立全局固定优先级调度中任务具有有限响应时间的一般性条件是一个主要的难点。

3.2　限制截止期任务的响应时间分析

本节考虑对限制截止期任务集的响应时间分析，即假设每一个任务τ_i的相对截止期D_i都小于等于其周期T_i。如上一节中所述，虽然[BC-RTA]是现有的固定优先级全局调度分析方法中性能最好的方法，但其分析精确度依旧比较差，分析结果依旧比较悲观。本章将提出一种新的固定优先级全局调度响应时间分析方法[NEW-RTA]，来大幅度地提高分析的精确度。多处理机全局调度中的关键时刻是未知的，换句话说，同步释放所有高优先级的任务未必导致被分析任务的最大响应时间。正是由于“关键时刻”是未知的，[BC-RTA]采用了一个悲观但是安全的近似来计算每个高优先级任务的干扰，因而导致了非常悲观的分析结果。[NEW-RTA]方法的关键思想在于使用一个与单处理机调度分析中关键时刻相类似的概念，本文称之为近似关键时刻。与关键时刻不同，近似关键时刻并非一个具体的任务释放序列，而是代表一组不同的任务释放序列的某些共有特性。通过利用近似关键时刻的性质，[NEW-RTA]对高优先级任务干扰使的估计更加精确，从而获得更加精确的分析结果。

3.2.1 [NEW-RTA]的总体框架

与单处理机固定优先级调度分析一样，对多处理机全局固定优先级调度的分析也可以根据任务的优先级顺序对各个任务逐一进行分析。这是因为在固定优先级调度中，低优先级任务的行为对高优先级任务不会产生任何影响。因此下文中只介绍[NEW-RTA]如何对某一个特定任务τ_k进行响应时间分析。

假设J_k是任务τ_k所释放的任意一个实例，并计算J_k响应时间的一个上限R_k。由于J_k是一个任意选择的实例，该响应时间上限R_k即为任务τ_k的响应时间上限。J_k的释放时间为r_k，完成时间为f_k。为了简化叙述，把空闲处理器看作是处理器正在执行一个优先级最低的空闲任务。

定义 3.1（$k-$**忙碌期**）：时间区域$\left[t_0, f_k\right)$为$k-$忙碌期，其中t_0为同时满足下列所有条件的时间点：

1. $t_0 < r_k$；
2. $\forall t \in [t_0, r_k]$:在t时刻所有的处理器都在执行比τ_k优先级高的任务；
3. $\forall t < t_0, \exists t' \in [t, t_0]$:在$t'$时刻至少有一个处理器在执行优先级不高于$\tau_k$的任务。

如果不存在满足上述所有条件的t_0，则令$t_0 = r_k$。

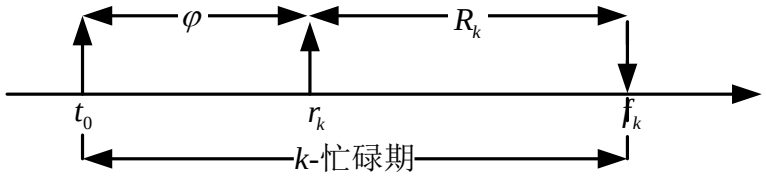


图 3.2 限制截止期任务的扩展 $k-$忙碌期示意图

Fig. 3.2 Illustration of extended level-k busy period for constrained-deadline task sets

图 3.2 是 $k-$忙碌期的示意图， 其中 $\varphi = r_k - t_0$。需要注意的是，$k-$忙碌期只是一个抽象的概念而非一个具体的已知时间区域，其中 f_k 为期望求得的任务结束时间（通过它根据 $R_k = f_k - r_k$ 立即可知 J_k 的响应时间），而 t_0 为一个未知的时间点。本文提出的响应时间分析是围绕 $k-$忙碌期进行的。

首选介绍如下引理：

引理 3.1： 在 $k-$忙碌期 $[t_0, f_k)$ 内，比 τ_k 优先级高的任务中最多 $M-1$ 个任务有前部任务实例。

证明：使用反证法证明。假设在 $k-$忙碌期 $[t_0, f_k)$ 内多于 $M-1$ 个比 τ_k 优先级高的任务有 carry-in，也就是说，至少有 M 个比 τ_k 优先级高的任务实例在 t_0 时刻还没有完成。令 t_x 为这 M 个高优先级任务实例的最晚释放时间，则可知 $t_x < t_0$ 且在时间区域 $[t_x, t_0]$ 内执行的所有任务的优先级都比 τ_k 高。根据 t_0 是否等于 r_k 分两种情况讨论：

（1）$t_0 \neq r_k$。根据 t_0 的定义可知 t_0 满足定义 3.1 中的三个条件。依上面讨论可知 $[t_x, t_0]$ 内执行的所有任务的优先级都比 τ_k 高，这与定义 3.1 中的第三个条件相矛盾。

（2）$t_0 = r_k$。根据 t_0 的定义可知，在 r_k 之前不存在同时满足定义 3.1 中条件 2 与 3 的时间点。根据上面讨论可知 t_x 满足条件 2，因此 t_x 一定不满足条件 3，即 t_x 满足 $\forall t < t_x$：在 t 时刻所有处理器在执行优先级高于 τ_k 的任务。然而当系统开始运行之前，所有处理器都处于空闲状态，也就是说 t_x 之前必然存在一点某个处理器在执行优先级最低的空闲任务，因此与上述讨论所知 t_x 的性质相矛盾。

综上所述，无论 t_0 是否等于 r_k，都将导致矛盾，因此假设不成立。

证毕。

通过上述引理可以看出，与[BC-RTA]中假设每个一个高优先级任务都有前部任务实例相比，$k-$忙碌期内有前部任务实例的高优先级任务数大大的减少了。但是使用 $k-$忙碌期带来的问题是并不知道 t_0 具体在哪个时间点，也就是说 $\varphi = r_k - t_0$ 是一个未知变量。文献[34]中提出以下办法来解决此问题：首先计算出所有需要考虑的 φ 值的一个上限值

Φ，然后让φ遍历$[0,\Phi]$中的每一个值来进行可调度性测试。只有φ取$[0,\Phi]$中的每一个值时都可通过可调度性测试，才判定该任务为可调度。

这种方法的问题在于，其将带来极高的计算复杂度。首先，φ的上限Φ是伪多项式复杂度，且实际中通常是一个比较大的值。尤其对于那些参数值比较大以及系统利用率比较大的任务集，Φ将是一个非常大的值。其次，响应时间分析过程本身也是伪多项式复杂度的。因此，如果直接使用[34]中的方法来直接遍历$[0,\Phi]$中的每一个φ值逐一进行响应时间分析，将导致非常高的复杂度，以致在实际应用中无法在可承受的时间内得到分析结果。

本章提出的响应时间分析方法将解决上述问题。将证明，为了计算J_k的最坏响应时间只需考虑$\varphi=0$，也就是t_0=r_k这一种情况。也就是说，使用$\varphi=0$所得的的响应时间上限对于任意其它的φ值都是安全的。

3.2.2 工作量与干涉

在介绍响应时间分析之前，先介绍在一定长度的时间区域内工作量（Workload）与干涉（Interference）的概念。

（1）工作量

一个任务在$k-$忙碌期内的工作量是这个任务在该时间区域内所有执行时间的累加。用$W_k(\tau_i,x)$来表示一个优先级比τ_i高的任务τ_i在长度为x的$k-$忙碌期内工作量上限。根据引理 3.1 可知最多有$M-1$个任务有前部任务实例，而其它的任务都没有前部任务实例。因此进一步定义以下两种不同工作量上限：

- $W_k^{NC}(\tau_i,x)$：如果τ_i没有前部任务实例，则$W_k^{NC}(\tau_i,x)$表示τ_i在长度为x的$k-$忙碌期内的工作量上限。
- $W_k^{CI}(\tau_i,x)$：如果τ_i有前部任务实例，则$W_k^{CI}(\tau_i,x)$表示τ_i在长度为x的$k-$忙碌期内的工作量上限。

可以通过以下引理计算$W_k^{NC}(\tau_i,x)$与$W_k^{CI}(\tau_i,x)$：

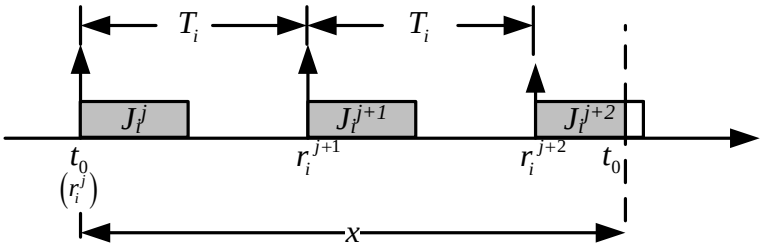
引理 3.2：一个高优先级任务τ_i在长度为x的$k-$忙碌期内工作量上限可以根据如下公式进行计算：

$$W_k^{NC}(\tau_i,x)=\left\lfloor \frac{x}{T_i} \right\rfloor \cdot C_i + \left(x \mod T_i \right)^{C_i} \tag{3.5}$$

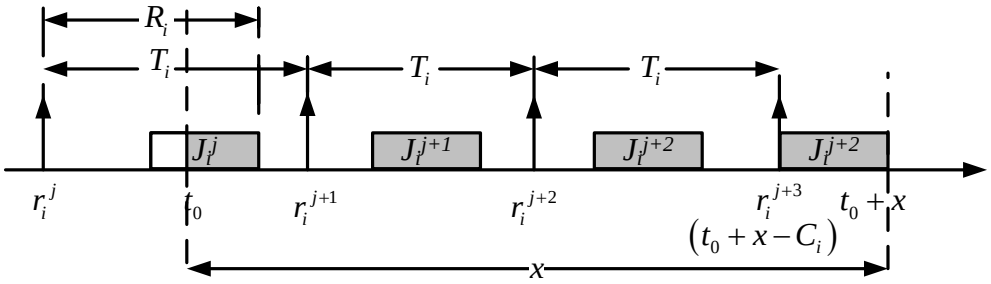
$$W_k^{CI}(\tau_i,x)=\left\lfloor \frac{x-C_i}{T_i} \right\rfloor \cdot C_i + C_i + \alpha \tag{3.6}$$

其中 $\alpha = \left\lfloor \left(x - C_i \right)_0 \bmod T_i - \left(T_i - R_i \right) \right\rfloor_0^{C_i}$。

证明：首先考虑 $W_k^{NC}\left(\tau_i, x\right)$。根据[40]可知，一个没有前部任务实例的任务 τ_i 当满足下列所有条件时，其在长度为 x 的 k – 忙碌期内的工作量为最大：（1）τ_i 的每个实例都执行其最坏情况执行时间；（2）τ_i 的所有实例以最小释放间隔 T_i 释放；（3）τ_i 某个实例的释放时间与 k – 忙碌期的起始时间同步；（4）τ_i 的后部任务实例在释放后立即执行。如图 3.3-(a)所示。



(a) $W_k^{NC}\left(\tau_i, x\right)$ 的计算
(a) Computation of $W_k^{NC}\left(\tau_i, x\right)$



(b) $W_k^{CI}\left(\tau_i, x\right)$ 的计算
(b) Computation of $W_k^{CI}\left(\tau_i, x\right)$

图 3.3 $W_k^{NC}\left(\tau_i, x\right)$ 和 $W_k^{CI}\left(\tau_i, x\right)$ 的计算示意图
Fig. 3.3 Illustration of the computation of $W_k^{NC}\left(\tau_i, x\right)$ and $W_k^{CI}\left(\tau_i, x\right)$

然后考虑 $W_k^{CI}\left(\tau_i, x\right)$。根据文献[40]的结论可知，一个具有前部任务实例的任务 τ_i 当满足下列所有条件时，其在长度为 x 的 k – 忙碌期内的工作量为最大：（1）τ_i 的每个实例都执行其最坏情况执行时间；（2）τ_i 的所有实例以最小释放间隔 T_i 释放；（3）τ_i 的前部任务实例所有的工作尽可能晚地执行；（4）τ_i 的后部任务实例实例在释放后立即不受干扰地执行至完成；（5）τ_i 的后部任务实例的执行结束时间与 k – 忙碌期的结束时间对齐，如图 3.3-(b) 所示。

□

引理 3.2 中 $W_k^{CI}\left(\tau_i, x\right)$ 的计算需要知道高优先级任务 τ_i 的响应时间上限 R_i。因为可以

按照由高至低的优先级顺序逐一地对各个任务进行响应时间分析，因此计算$W_k^{CI}(\tau_i,x)$时，即当分析τ_k时，已经知道每个高优先级任务响应时间的上限。

值得注意的是，引理 3.2 中$W_k^{NC}(\tau_i,x)$ 与 $W_k^{CI}(\tau_i,x)$ 的计算都不依赖于φ。这意味着，只要给定了$k-$忙碌期的长度x，无论t_0具体在哪个时刻（也就是说无论φ为多大），永远得到相同的$W_k^{NC}(\tau_i,x)$ 和 $W_k^{CI}(\tau_i,x)$计算结果。这一性质是本章提出的响应时间方法不需要遍历所有的φ值的关键。

（2）干涉

在工作量的基础上，可以进而定义每个高优先级任务在长度为x 的$k-$忙碌期内的干涉时间上限$I_k(\tau_i,x)$。一个任务τ_i的干涉时间代表着其工作量中实际对τ_k产生干涉的部分，也就是实际阻止了τ_k执行的部分。工作量与干涉时间的区别主要在于，高优先级任务τ_i的工作量中可能有一部分是与τ_k的并行执行的，而τ_i的这一部分工作量实际上没有阻止τ_k的执行。所有一般来讲，一个任务在$k-$忙碌期内的干涉时间小于其工作量，因此在响应时间分析中使用任务的干涉时间会得到比使用任务的工作量更加精确的分析结果。与工作量类似，同样定义两种类型的干涉时间上限：

- $I_k^{NC}(\tau_i,x)$：如果τ_i没有前部任务实例，则$W_k^{NC}(\tau_i,x)$表示τ_i在长度为x的$k-$忙碌期内的干涉时间上限。

$$I_k^{NC}(\tau_i,x)=\left\lfloor W_k^{NC}(\tau_i,x)\right\rfloor^{x-C_k} \tag{3.7}$$

- $I_k^{CI}(\tau_i,x)$：如果τ_i有前部任务实例，则$W_k^{CI}(\tau_i,x)$表示τ_i在长度为x的$k-$忙碌期内的干涉时间上限。

$$I_k^{CI}(\tau_i,x)=\left\lfloor W_k^{CI}(\tau_i,x)\right\rfloor^{x-C_k} \tag{3.8}$$

现在定义长度为x的$k-$忙碌期内的干涉时间总和上限$\Omega_k(x)$：

$$\Omega_k(x)=\max_{(\tau^{NC},\tau^{CI})\in Z}\left(\sum_{\tau_i\in\tau^{NC}}I_k^{NC}(\tau_i,x)+\sum_{\tau_i\in\tau^{CI}}I_k^{CI}(\tau_i,x)\right) \tag{3.9}$$

将所有高优先级任务的集合$hp(\tau_k)$划分成满足下列条件的两个子集τ^{NC}与τ^{CI}：

1. $\tau^{NC}\cup\tau^{CI}=hp(\tau_k)$,
2. $\tau^{NC}\cap\tau^{CI}=\phi$
3. $|\tau^{CI}|\le M-1$

$Z\subseteq\tau\times\tau$表示对τ^{NC} 与 τ^{CI}所有可能划分的集合。通过取所有可能的划分中所得的最大值，$\Omega_k(x)$代表了当最多$M-1$个高优先级任务有前部任务实例而其它高优先级任

务没有前部任务实例时，所有高优先级任务在长度为x的$k-$忙碌期内干涉时间总和上限的最大值。根据$\Omega_k(x)$的定义，其将取所有τ^{NC}与τ^{CI}可能的组合中的最大值。然而在实际计算$\Omega_k(x)$时，并不需要遍历所有τ^{NC}与τ^{CI}组合的可能性。其计算方法如图 3.4 所示：

1：　$\forall \tau_i : I_k^{diff}(\tau_i, x) = I_k^{CI}(\tau_i, x) - I_k^{NC}(\tau_i, x)$

2：　$sum_1 = \sum_{\tau_i \in \tau} I_k^{NC}(\tau_i, x)$

3：　$sum_2 =$所有任务中最大的$M-1$个$I_k^{diff}(\tau_i, x)$之和

4：　$\Omega_k(x) = sum_1 + sum_2$

图 3.4 计算干涉总和上限算法

Fig. 3.4 The algorithm of computing the total interference upper bound

在图 3.4 的算法中，计算所有任务的$I_k^{diff}(\tau_i, x)$为线性复杂度，计算sum_1也为线性复杂度。计算sum_2可以使用线性选择算法[120]来找到所有$I_k^{diff}(\tau_i, x)$中最大的 M-1 个。因此图 3.4 算法的复杂度为$O(N)$。

引理 3.3：下述条件对于任意$x < f_k - t_0$都成立*:*

$$\left\lfloor \frac{\Omega_k(x)}{M} \right\rfloor > x - C_k \tag{3.10}$$

证明：根据$\Omega_k(x)$的定义可知$\Omega_k(x)$是取所有可能的τ^{NC} 与 τ^{CI}划分的最大值。将导致$\Omega_k(x)$最大值的划分记为$\tau^{NC'}$与$\tau^{CI'}$，则有：

$$\Omega_k(x) = \sum_{\tau_i \in \tau^{CI}} I_k^{CI}(\tau_i, x, h) + \sum_{\tau_i \in \tau^{NC}} I_k^{NC}(\tau_i, x, h)$$

令$\vartheta^{CI} \subseteq \tau^{CI'}$ 和 $\vartheta^{NC} \subseteq \tau^{NC'}$为满足下列条件的集合：

$$\forall \tau_i \in \vartheta^{CI} : W_k^{CI}(\tau_i, x) > x - C_k$$

$$\forall \tau_i \in \vartheta^{NC} : W_k^{NC}(\tau_i, x) > x - C_k$$

因此$\vartheta = \vartheta^{CI} \cup \vartheta^{NC}$ 代表了所有高优先级中那些在$k-$忙碌期内最坏情况的工作量非常大而不得不与J_k并行执行的任务的集合。于是可以将$\Omega_k(x)$的定义重写成如下形式：

$$\Omega_k(x) = |\vartheta| \cdot (x - C_k) + \sum_{\tau_i \in \tau^{CI} \setminus \vartheta^{CI}} W_k^{CI}(\tau_i, x) + \sum_{\tau_i \in \tau^{NC} \setminus \vartheta^{NC}} W_k^{NC}(\tau_i, x) \tag{3.11}$$

其中$|\vartheta|$是ϑ中任务的个数。根据$|\vartheta|$是否大于M分两种情况进行讨论：

1. $|\vartheta| > M$。这种情况下有

$$\left\lfloor \frac{\Omega_k(x)}{M} \right\rfloor > \left\lfloor \frac{M(x-C_k)}{M} \right\rfloor \geq x-C_k$$

所以定理成立。

2. $|\vartheta| \leq M$。令 $x < f_k - t_0$，因为 f_k 为 J_k 的完成时间，可知 J_k 在时间点 $t_0 + x$ 依旧是活跃实例。因此在时间区域 $[t_0, t_0 + x)$ 内 J_k 的执行时间严格小于 C_k。因为 $[t_0, t_0 + x)$ 内的每个时间点上 ϑ 中的任务最多占用 $|\vartheta|$ 个处理器，所以其余的任务（$\tau \setminus \vartheta$ 中的任务）累计占用其余的 $M - |\vartheta|$ 个处理器的时间至少为 $x - C_k$（否则的话 J_k 将在 $t_0 + x$ 时间点前执行 C_k 并完成执行，这与 $x < f_k - t_0$ 相矛盾）。因此，$\tau \setminus \vartheta$ 中的任务在 $[t_0, t_0 + x)$ 内所产生的工作量至少为 $(M - |\vartheta|) \cdot (x - C_k)$。又因为 $W_k^{CI}(\tau_i, x)$ 和 $W_k^{NC}(\tau_i, x)$ 是任务工作量的上限，因此可知

$$\sum_{\tau_i \in \tau^{CI} \setminus \vartheta^{CI}} W_k^{CI}(\tau_i, x) + \sum_{\tau_i \in \tau^{NC} \setminus \vartheta^{NC}} W_k^{NC}(\tau_i, x) > (M - |\vartheta|) \cdot (x - C_k) \tag{3.12}$$

根据(3.11)与(3.12)可知 $\Omega_k(x) > M \cdot (x - C_i)$，据此引理可证。 □

引理 3.3 的直观意义如下：如果假设 $k-$忙碌期在 J_k 的完成时间之前结束，那么这期间所有高优先级任务干涉时间的总和将会足够使 J_k 以及高优先级任务无法在 $k-$忙碌期之内全部完成。这也就意味着实际上 $k-$忙碌期实际还没有结束，而可以继续向后扩展。根据这个性质可以在下一章中使用迭代的计算过程来进行响应时间分析。

3.2.3 响应时间迭代分析过程

$k-$忙碌期起始于时间点 t_0，即比 J_k 的释放时间 r_k 提前了时间 φ。φ 是一个未知变量，但是暂时先假设给定了一个特定的 φ 值，并讨论如何在这个特定的 φ 值下求得 J_k 的响应时间上限。

引理 3.4：给定一个 $\varphi \geq 0$，令 χ 为下述方程式的最小解：

$$x = \left\lfloor \frac{\Omega_k(x)}{M} \right\rfloor + C_k \tag{3.13}$$

则 $\chi - \varphi$ 是当 $k-$忙碌期起点为 $t_0 = r_k - \varphi$ 时 τ_k 的响应时间上限。

证明：用反正法证明。令 J_k 为 τ_k 响应时间最大的实例，且 J_k 在 $t_0 = r_k - \varphi$ 的情况下的响应时间为 R，并且假设

$$\chi - \varphi < R \tag{3.14}$$

R 是 J_k 响应时间，即存在一个 τ 的任务释放序列使 J_k 满足

$$f_k - r_k = R \tag{3.15}$$

根据(3.14)和(3.15)可知 $\chi < f_k - t_0$。又根据引理3.3可知

$$\left\lfloor \frac{\Omega_k(\chi)}{M} \right\rfloor > \chi - C_k$$

这与 χ 为方程式(3.13)的最小解相矛盾，据此引理可证。　　□

到目前为止，已经知道如何在给定 φ 值得情况下求得 τ_k 的响应时间上限。由于 φ 是一个未知量，现在暂时还不知道在没有给定 φ 值的情况下，如何求得 τ_k 的响应时间上限。解决这个问题的办法之一是找到 φ 值的上限 Φ，并遍历 $[0,\Phi]$ 中的每个值来求解响应的响应时间上限，最后取所有 φ 值下响应时间上限的最大值，即为 τ_k 的响应时间上限值。但是由于 φ 值的上限 Φ 是一个非常大的值，使用这种方法将导致非常高的时间复杂度（伪多项式复杂度）。下面将证明，为了获取安全的 τ_k 响应时间上限，只需要考虑 $t_0 = r_k$（即 $\varphi = 0$）一种情况。

定理 3.1：令 χ 为如下方程式的最小解

$$x = \left\lfloor \frac{\Omega_k(x)}{M} \right\rfloor + C_k \tag{3.16}$$

则 χ 为 τ_k 的最坏响应时间上限。

证明：$W_k^{NC}(\tau_i, x)$ 与 $W_k^{CI}(\tau_i, x)$ 与 φ 相独立，进而 $\Omega_k(x)$ 也与 φ 相独立。因此无论 φ 的值为多少，方程式(3.16)的最小不动点都相同。根据引理 4 可知，对于一个给定的 φ 值，$\chi - \varphi$ 是 τ_k 的响应时间上限。因此，在所有的 $\varphi \in [0,\Phi]$ 值中，当 $\varphi = 0$ 时 $\chi - \varphi$ 值为最大，即 χ 为 τ_k 的最坏响应时间上限。　　□

定理 3.1 可以被看成在全局固定优先级多处理机调度中的一种与“关键时刻”相类似的概念。在定理 3.1 提供的响应时间分析中，$\varphi = 0$ 是所有可能中的最坏情况。也就是说，所有处理器在 J_k 的释放时间当开始执行高优先级任务会导致 J_k 的最坏响应时间。在单处理机固定优先级调度中，该情况等价于所有高优先级任务都与 J_k 同步释放，这正是所谓的“关键时刻”，即导致最坏响应时间任务释放序列。在全局固定优先级多处理机调度中，该情况并不是一个具体的最坏情况任务释放序列，但是它可以被看作是若干任务释放序列的集合。

与单处理机调度中经典响应时间分析一样，定理 1 的分析可以通过迭代的方法实现：用 x^i 表示第 i 步迭代得到的方程左边 x 值。令 $x^0 = C_k$，因为知道 τ_k 的最坏响应时间一定不会小于 C_k。使用下列等式进行迭代：

$$x^i = \left\lfloor \frac{\Omega_k(x^{i-1})}{M} \right\rfloor + C_k$$

迭代过程直至 $x^i = x^{i-1}$ 或 $x^i > D_k$ 停止。如果因 $x^i = x^{i-1}$ 停止，则 τ_k 的最坏响应时间上

限 χ 为 x^i，且知 τ_k 可调度；如果因 $x^i > D_k$ 停止，则 τ_k 的最坏响应时间上限超过其截止期，因此判定 τ_k 为可调度。

则 χ 为 τ_k 的最坏响应时间上限。

推论 3.5：定理3.1中的响应时间分析方法优于[BC-RTA]。

证明：[BC-RTA]使用如下迭代方程式求解响应时间上限：

$$x = \left\lfloor \frac{\sum_{\tau_i < hp(k)} I_k(\tau_i, x)}{M} \right\rfloor + C_k \tag{3.17}$$

根据 $\Omega_k(x)$ 的定义可知

$$\Omega_k(x) \le \sum_{\tau_i < hp(k)} I_k(\tau_i, x)$$

因此方程式(3.16)的右侧小于等于方程式(3.17)的右侧，所以方程式(3.16)的最小解不大于方程式(3.17)的最小解，即根据定理 1 所得到的响应时间上限不大于根据 BC-RTA 得到的响应时间上限。 □

3.3　任意截止期任务的响应时间分析

本节将定理 3.1 的响应时间分析方法扩展到任意截止期任务集，即考虑被分析任务 τ_k 的相对截止期 D_k 大于其周期 T_k 的情况。

令 J_k^1 为 τ_k 任意一个满足条件 $\gamma_k^1 = r_k^1$ 的实例，也就是说当 J_k^1 释放时，之前释放的所有实例都已完成。值得注意的是一定存在一个这样的实例 J_k^1，因为 τ_k 释放的第一个实例即满足上述条件。

定义 3.2：定义 $[t_0, f_k^H]$ 为 $k-$忙碌期，其中 t_0 为 r_k^1 之前满足下述条件的时间点：

- $t_0 < r_k$；
- $\forall t \in [t_0, r_k]$：在 t 时刻所有的处理器都在执行比 τ_k 优先级高的任务；
- $\forall t < t_0, \exists t' \in [t, t_0]$：在 t' 时刻至少有一个处理器在执行优先级不高于 τ_k 的任务。

如果不存在满足上述所有条件的 t_0，则令 $t_0 = r_k^1$。令 J_k^H（$H \ge 1$）为 J_k^1 或 J_k^1 之后第一个响应时间不超过周期 T_k 的实例，f_k^H 为 J_k^H 的执行结束时间。

与上一章中相似，定义 $\varphi = r_k^1 - t_0$。图 3.5 中所示为新的 $k-$忙碌期。这里暂且假设存在一个符合定义 3.2 的 J_k^H。在本节最后中将讨论在何种条件下才能保证存在这样一个 J_k^H。

根据上面 t_0 的定义可知上一章中的引理 3.1 对于新的 $k-$忙碌期依旧成立，也就是在

$k-$忙碌期$[t_0, f_k)$内，最多有$M-1$个高优先级任务有前部任务实例。

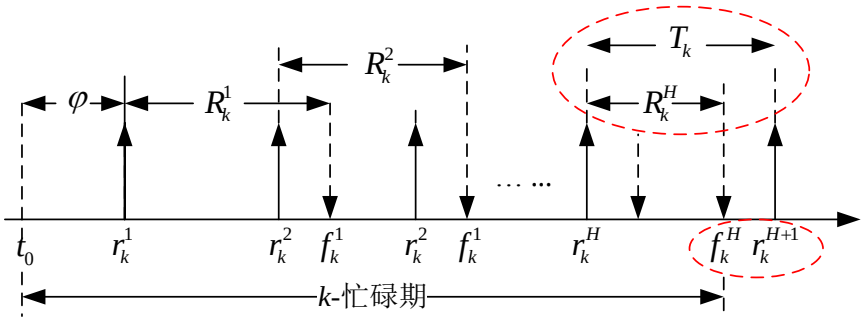


图 3.5 任意截止期任务在的扩展$k-$忙碌期示意图

Fig. 3.5 Illustration of extended level-k busy period for arbitrary-deadline task sets

3.3.1 工作量与干涉

（1）工作量

对于任意截止期任务，其$W_k^{NC}(\tau_i, x)$ 的计算与限制截至期任务的情况相同。对于$W_k^{CI}(\tau_i, x)$的计算，与限制截至期任务的区别在于任意截止期任务高优先级任务τ_i 的响应时间R_i有可能大于其周期T_i。下面介绍如何计算一个任意截止期任务的$W_k^{CI}(\tau_i, x)$

$W_k^{CI}(\tau_i, x)$由三部分组成：（i）前部任务实例的工作量，（ii）中部任务实例的工作量，（iii）后部任务实例的工作量。其中第（ii）与（iii）项的计算与限制截止期任务的情况相同，分别为$\left\lfloor \frac{x-C_i}{T_i} \right\rfloor \cdot C_i$和$C_i$。下面集中考虑第（i）项的计算。

图 3.6 中示意了前部任务工作量的计算。第一步是确定第一个中部任务实例的释放时间与t_0之间的距离，记为l_1，计算公式如下：

$$l_1 = (x - C_i) \bmod T_i$$

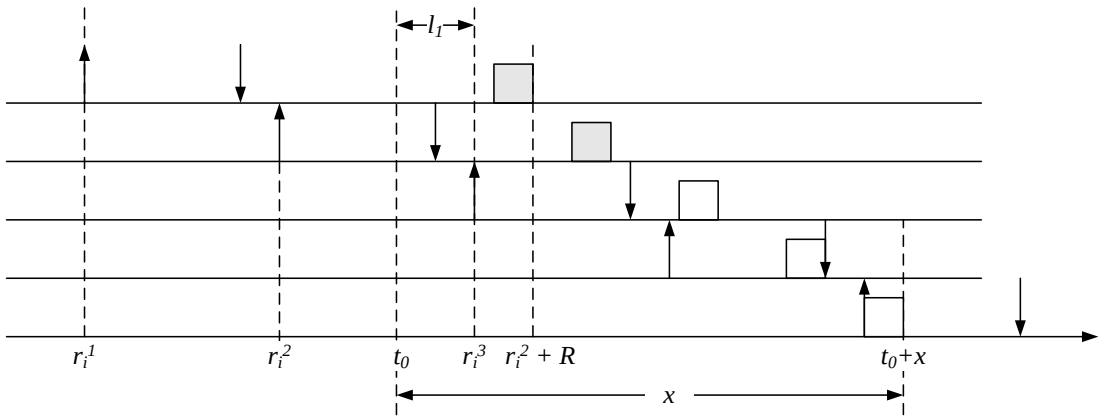


图 3.6 任意截止期任务$W_k^{CI}(\tau_i, x)$中前部任务工作量计算意图

Fig. 3.6 Illustration of computation of the carry-in workload in $W_k^{CI}(\tau_i, x)$ for arbitrary-deadline tasks

下一步是计算有多少个前部任务实例的工作可能在$[t_0,t_0+x)$间执行。假设J_i^j为第一个中部任务实例，J_i^{j-y}为第一个可能在$[t_0,t_0+x)$间执行的任务实例。若使J_k^{i-y}可能在$[t_0,t_0+x)$间执行，则其比满足$f_i^{j-y}>t_0$，即

$$r_i^j - y\times T_i + R_i > t_0$$

根据上面所述 l1 的计算可得

$$x - C_i\ {}_0 \bmod T_i + R_i > y\times T_i$$

因为y为最大的满足上式的整数，可知：

$$y = \left\lfloor \frac{x - C_i\ {}_0 \bmod T_i + R_i}{T_i} \right\rfloor$$

也就是说，最多可能有$y = \left\lfloor \frac{x - C_i\ {}_0 \bmod T_i + R_i}{T_i} \right\rfloor$个前部任务实例。从第二个前部任务实例起，每个任务实例贡献的工作量上限为C_i。第一个前部任务实例贡献的工作量除了不能高于上限C_i，同时还不能高于$f_i^{j-y}-t = \left[l_1 - y\times T_i + R_i \right]_0$。根据上述 l1 与 y 的计算可知：

$$f_i^{j-y} - t = \left[x - C_i\ {}_0 \bmod T_i - \left\lfloor \frac{x - C_i\ {}_0 \bmod T_i + R_i}{T_i} \right\rfloor \times T_i + R_i \right]_0$$

因此，所有前部任务实例的工作量总和为：

$$\beta = \left[y - 1\ {}_0 \times C_i + \left[x - C_i\ {}_0 \bmod T_i - \left\lfloor \frac{x - C_i\ {}_0 \bmod T_i + R_i}{T_i} \right\rfloor \times T_i + R_i \right]_0^{C_i} \right]_0$$

综上所述，一个任意截止期任务的$W_k^{CI}(\tau_i,x)$计算如下：

$$W_k^{CI}(\tau_i,x) = \left\lfloor \frac{x - C_i\ {}_0}{T_i} \right\rfloor \cdot C_i + C_i + \beta$$

（2）干涉

与限制截止期任务类似，一个任意截止期任务的干涉时间为其工作量减掉其中必然与中τ_k并行执行的部分。与限制截止期任务不同的是，在扩展的$k-$忙碌期内可能包含若干个τ_k的任务实例。因此干涉时间中引入一个新的参数h，代表$k-$忙碌期内所包含τ_k任务实例的个数。同样定义两种类型的干涉时间上限：

$$I_k^{NC}(\tau_i,x,h) = \left[W_k^{NC}(\tau_i,x) \right]_0^{x-h\cdot C_k+1}$$

$$I_k^{CI}(\tau_i,x,h) = \left[W_k^{CI}(\tau_i,x) \right]_0^{x-h\cdot C_k+1}$$

同时为干涉时间总和上限也引入一个新的参数 h：

$$\Omega_k(x,h)=\max_{(\tau^{NC},\tau^{CI})\in Z}\left(\sum_{\tau_i\in\tau^{NC}}\mathrm{I}_k^{NC}(\tau_i,x,h)+\sum_{\tau_i\in\tau^{CI}}\mathrm{I}_k^{CI}(\tau_i,x,h)\right) \tag{3.18}$$

3.3.2 响应时间迭代分析过程

这一小节将把第 3.3.1 节中的响应时间分析过程扩展到任意截止期的情况。与第 3.3.1 节类似，首先假设一个特点的 φ，即下述分析首先考虑一个特定的 $t_0=r_k^1-\varphi$。首先将引理 3 和引理 4 推广到任意截止期任务的情况：

引理 3.6：下述条件对于任意 $x<f_k-t_0$ 都成立:

$$\left\lfloor\frac{\Omega_k(x,h)}{M}\right\rfloor>x-h\cdot C_k$$

引理 3.6 的证明方法与引理 3.3 类似。

引理 3.7：给定一个 $t_0=r_k^1-\varphi$，对于任意 $h\geq 1$ 令 χ^h 为以下方程的最小解：

$$x=\left\lfloor\frac{\Omega_k(x,h)}{M}\right\rfloor+h\cdot C_k \tag{3.19}$$

令 $H(\varphi)$ 为满足以下不等式的最小值

$$\chi^{H(\varphi)}\leq H(\varphi)\cdot T_k+\varphi \tag{3.20}$$

那么

$$R_k^{\varphi}=\max_{h\in[1,H(\varphi)]}\left\{\chi^h-(h-1)\cdot T_k-\varphi\right\}$$

为此特定 t_0 点下 τ_k 的一个响应时间上限。

证明方法与引理 3.4 类似。

与第 3.3.1 节中限制截止期的情况类似，上述内容可以得到一个特点的 φ 下 τ_k 的响应时间上限。但是 φ 是一个未知量，因此，为了得到 τ_k 的一个安全的响应时间上限，必须计算所有可能的 φ 取值下的 R_k^{φ}，并取其中的最大值。

实际上，不必计算每个 φ 取值下的 R_k^{φ} 就可以获得 τ_k 的安全的响应时间上限：只需要计算 $\varphi=0$ 这一种情况下的 R_k^{φ}。

对于任意截止期任务，得到这个结论要比第 3.3.1 节中对限制截止期的讨论更复杂一些，原因是引理 3.7 中需要处理一组，而非一个方程的解。因此，对于 $\varphi_1>\varphi_2$，无法像第 3.3.1 节中那样立即得到 $R_k^{\varphi_1}$ 与 $R_k^{\varphi_2}$ 之间的关系。

下面来说明为何对于任意截止期任务同一只需要考虑 $\varphi=0$ 这一种情况下的 R_k^{φ}。再次使用如下性质：$\Omega_k(x,h)$ 的值不依赖于 φ。因此，对于任意一个 h，方程(3.19)的最小

解不依赖于φ。所以根据不等式(3.20)可知，如果$\varphi_1 > \varphi_2$则必有$H(\varphi_1) \le H(\varphi_2)$（因为对于一个较大的$\varphi$，不等式(3.20)的约束更松弛）。令

$$S^{\varphi_1} = \left\{ \chi^h - (h-1)\cdot T_k - \varphi_1 \right\}_{h\in[1,H(\varphi_1)]}$$

$$S^{\varphi_2} = \left\{ \chi^h - (h-1)\cdot T_k - \varphi_2 \right\}_{h\in[1,H(\varphi_2)]}$$

根据上述讨论可知S^{φ_1}中元素个数小于S^{φ_2}中的元素个数。根据$\varphi_1 > \varphi_2$又可知S^{φ_1}中的第i个值小于S^{φ_2}中的第i个值。综上所述，可知S^{φ_1}中元素的最大值小于S^{φ_2}中的元素的最大值，因此可知：

$$\varphi_1 > \varphi_2 \Rightarrow R_k^{\varphi_1} < R_k^{\varphi_2}$$

所以当$\varphi = 0$时R_k^{φ}得到最大值，因此只需要考虑$\varphi = 0$一种情况便可得到τ_k的响应时间上限。

在最终介绍任意截止期任务的响应时间分析方法之前，先介绍两个谓词作为响应时间分析的终止条件：

- 任务实例在下一个周期开始前执行完毕：

$$Term(h,k) \equiv \left[\chi^h \le h\cdot T_k \right]$$

- 任务实例错过截止期：

$$Miss(h,k) \equiv \left[\chi^h > (h-1)\cdot T_k + D_k \right]$$

定理 3.2：对于每个$h \ge 1$，令χ^h为下述方程的最小解。该最小解从初始值$x = h\cdot C_k$开始迭代求得。

$$x = \left\lfloor \frac{\Omega_k(x,h)}{M} \right\rfloor + h\cdot C_k \tag{3.21}$$

令 H 为满足$Term(h,k)$的最小 h 值，则

$$R_k = \max_{h\in[1,H]} \left\{ \chi^h - (h-1)\cdot T_k \right\}$$

是τ_k的一个响应时间上限。

证明：根据引理 5 和上述讨论可证。　□

上述定理中的响应时间分析过程可以通过依次对$h = 1,2,...$计算$x = \left\lfloor \frac{\Omega_k(x,h)}{M} \right\rfloor + h\cdot C_k$的最小解，直到满足终止条件$Term(h,k)$或者$Miss(h,k)$。如果计

算因$Term(h,k)$被满足而终止，则判定任务τ_k为可调度，否则判断其为不可调度。

3.3.3 分析过程的终止条件

上小节中介绍的响应时间分析过程如果因满足$Term(h,k)$而终止，则可获得τ_k的一个安全响应时间上限。下面将要证明，条件$Term(h,k)$在某些情况下一定会被满足。而在其它情况下，另一个终止条件$Miss(h,k)$会最终被满足。需要注意的是，现已知在解一个方程(3.21)时的不动点计算一定会终止，下面需要讨论的是整个响应时间计算过程是否会终止（整个过程需要多次进行解方程(3.21)的不动点计算）。

首先定义一个新的概念，τ_i的相对于τ_k的干涉利用率：

$$V_i^k = \min(U_i, 1-U_k) \tag{3.22}$$

干涉利用率V_i^k可以被看作利用率U_i在多处理机调度中的扩展。直观上讲，它限制为τ_i的利用率中与τ_k不能并行执行的部分。使用这个概念，下面来证明定理 3.2 中响应时间分析过程的终止条件。

定理 3.3：对于满足下述条件的任意一个任务

$$\sum_{i<k} V_i^k + M \cdot U_k \neq M \tag{3.23}$$

定理 3.2 中的响应时间分析过程一定会终止，即下述条件一定成立：

$$\exists h \geq 1 : Term(h,k) \vee Miss(h,k)$$

证明：根据不等式(3.22)左右两边的大小关系进行分类讨论：

第一种情况：

$$\sum_{i<k} V_i^k + M \cdot U_k < M \tag{3.24}$$

要证明必然存在一个$h \geq 1$使得终止条件$Term(h,k)$和$Miss(h,k)$中的至少一个成立。假设终止条件$Miss(h,k)$对于任意$h \geq 1$都不成立，也就是说方程(3.21)总是存在一个最小解。进一步假设对于任意$h \geq 1$终止条件$Term(h,k)$也都不成立，即$\forall h \geq 1 : \chi^h > h \cdot T_k$。挑选其中任意一个$h$，可知

$$\varepsilon := \chi^h - h \cdot T_k > 0 \tag{3.25}$$

证明的思路为，首先根据上式为$\Omega_k(\chi^h, h)$找到一个上限，然后以求得χ^h的上限。

因为h是任意选取，且$(\chi^h)_{h<1}$是一个严格递增的序列，这与求得的χ^h上限值相矛盾。

根据$\Omega_k(x,h)$的定义可得：

$$\Omega_k\left(\chi^h,h\right)\le\sum_{i<k}I_k^{CI}\left(\tau_i,\chi^h,h\right)$$

$$\Omega_k\left(\chi^h,h\right)\le\sum_{i<k}\min\left(W_k^{CI}\left(\tau_i,\chi^h\right),\chi^h-h\cdot C_k+1\right)$$

根据ε的定义可以知：$h\cdot C_k=\left(\chi^h-\varepsilon\right)\cdot U_k$。将其应用到上式可得

$$\Omega_k\left(\chi^h,h\right)\le\sum_{i<k}\min\left(W_k^{CI}\left(\tau_i,\chi^h\right),\chi^h-\left(\chi^h-\varepsilon\right)\cdot U_k+1\right)$$

又根据$W_k^{CI}\left(\tau_i,x\right)\le x\cdot U_i+2\cdot C_i$可得

$$\Omega_k\left(\chi^h,h\right)\le\sum_{i<k}\min\left(\chi^h\cdot U_i+2\cdot C_i,\chi^h-\left(\chi^h-\varepsilon\right)\cdot U_k+1\right)$$

$$\Omega_k\left(\chi^h,h\right)\le\sum_{i<k}\min\left(\chi^h\cdot U_i+2\cdot C_i,\chi^h\cdot\left(1-U_k\right)+\varepsilon\cdot U_k+1\right)$$

根据不等式性质$\min\left(a+b,c\right)\le\min\left(b,c\right)+a$，上式化为：

$$\Omega_k\left(\chi^h,h\right)\le\sum_{i<k}\min\left(\chi^h\cdot U_i,\chi^h\cdot\left(1-U_k\right)+\varepsilon\cdot U_k+1\right)+2\cdot\sum_{i<k}C_i$$

令η为满足$U_i>1-U_k$的任务个数。根据不等式性质

$$b\ge c\Rightarrow\min\left(a+b,c\right)\le\min\left(b,c\right)$$

上式可化为：

$$\Omega_k\left(\chi^h,h\right)\le\sum_{i<k}\min\left(\chi^h\cdot U_i,\chi^h\cdot\left(1-U_k\right)\right)+2\cdot\sum_{i<k}C_i+\left(\varepsilon\cdot U_k+1\right)\cdot\eta$$

根据(3.24)可知$\eta\le M-1$，因此得到：

$$\Omega_k\left(\chi^h,h\right)\le\chi^h\sum_{i<k}\min\left(U_i,1-U_k\right)+2\cdot\sum_{i<k}C_i+\left(\varepsilon\cdot U_k+1\right)\cdot\left(M-1\right)$$

$$\Omega_k\left(\chi^h,h\right)\le\chi^h\sum_{i<k}V_i^k+2\cdot\sum_{i<k}C_i+\left(\varepsilon\cdot U_k+1\right)\cdot\left(M-1\right)$$

由于$\varepsilon>0$,上式可化为：

$$\Omega_k\left(\chi^h,h\right)\le\chi^h\sum_{i<k}V_i^k+2\cdot\sum_{i<k}C_i+\left(\varepsilon\cdot U_k+1\right)\cdot M \tag{3.26}$$

下面将据此推出χ^h的一个上限。首先，因为χ^h是方程(3.21)的一个解，可知：

$$\chi^h = \left\lfloor \frac{\Omega_k\left(\chi^h, h\right)}{M} \right\rfloor + h \cdot C_k$$

$$\chi^h \le \frac{\Omega_k\left(\chi^h, h\right)}{M} + h \cdot C_k$$

将不等式(3.26)应用于上式可得：

$$\chi^h < \frac{\chi^h \sum_{i<k} V_i^k + 2 \cdot \sum_{i<k} C_i + \left(\varepsilon \cdot U_k + 1\right) \cdot M}{M} + h \cdot C_k$$

根据 ε 的定义可知 $\varepsilon \cdot U_k = \chi^h \cdot U_k - h \cdot C_k$，因此上式可化为：

$$\chi^h < \frac{\chi^h \sum_{i<k} V_i^k + 2 \cdot \sum_{i<k} C_i + \left(\chi^h \cdot U_k - h \cdot C_k + 1\right) \cdot M}{M} + h \cdot C_k$$

整理上式，使 χ^h 只在不等式的一边出现：

$$\chi^h \cdot \left(M - \sum_{i<k} V_i^k - M \cdot U_k \right) < 2 \cdot \sum_{i<k} C_i \cdot M$$

根据(3.24)可知 $M - \sum_{i<k} V_i^k - M \cdot U_k > 0$，将其应用于上式，最终可得：

$$\chi^h < \frac{2 \cdot \sum_{i<k} C_i + M}{M - \sum_{i<k} V_i^k - M \cdot U_k}$$

即存在一个 χ^h 的上限，这与 h 是任意选取且 $\left(\chi^h\right)_{h<1}$ 是一个严格递增的序列相矛盾，因此假设不成立，于是证得两个终止条件中必有一个被满足。

第二种情况 ：

$$\sum_{i<k} V_i^k + M \cdot U_k > M \tag{3.27}$$

要证明必然存在一个 $h \ge 1$ 使得终止条件 $Term(h, k)$ 和 $Miss(h,k)$ 中的至少一个成立，即 $\chi^h \le h \cdot T_k$ 与 $\chi^h > (h-1) \cdot T_k + D_k$ 中至少一个成立。如果 $D_k \le T_k$ 则显然成立，因此只需要考虑 $D_k > T_k$ 这一种情况。

使用反正法，假设上述两个条件对任意 $h \ge 1$ 都不成立，即

$$\forall h \ge 1 : \chi^h \in \left(h \cdot T_k, (h-1) \cdot T_k + D_k\right) \tag{3.28}$$

证明的思路为首先为 $\Omega_k\left(\chi^h, h\right)$ 找到一个上限，然后用其得到一个 χ^h 的上限。因为 h 是任意选择且 $\left(\chi^h\right)_{h\ge1}$ 是一个严格递增的序列，这与 χ^h 的上限值相矛盾。

根据 $\Omega_k\left(x, h\right)$ 的定义可以得到：

$$\Omega_k\left(\chi^h,h\right)\geq\sum_{i<k}\mathrm{I}_k^{NC}\left(\tau_i,\chi^h,h\right)$$

$$\Leftrightarrow\Omega_k\left(\chi^h,h\right)\geq\sum_{i<k}\min\left(W_k^{NC}\left(\tau_i,\chi^h\right),\chi^h-h\cdot C_k+1\right)$$

因为$W_k^{NC}\left(\tau_i,x\right)\geq x\cdot U_i$，上式化为：

$$\Omega_k\left(\chi^h,h\right)\geq\sum_{i<k}\min\left(\chi^h\cdot U_i,\chi^h-h\cdot C_k+1\right)$$

根据(3.28)可知 $\chi^h>h\cdot T_k$，因此得到$\chi^h-h\cdot C_k+1>\chi^h\left(1-U_k\right)+1$。将其应用于上式可以得到：

$$\Omega_k\left(\chi^h,h\right)\geq\sum_{i<k}\min\left(\chi^h\cdot U_i,\chi^h\left(1-U_k\right)\right)$$

根据V_i^k的定义上式可以化为：

$$\Omega_k\left(\chi^h,h\right)\geq\chi^h\sum_{i<k}V_i^k \tag{3.29}$$

下面将由此求出χ^h的一个上限。因为χ^h是方程(3.21)的一个解，可知：

$$\chi^h=\left\lfloor\frac{\Omega_k\left(\chi^h,h\right)}{M}\right\rfloor+h\cdot C_k>\frac{\Omega_k\left(\chi^h,h\right)}{M}+h\cdot C_k-1$$

应用不等式(3.29)到上式可得：

$$\chi^h<\frac{\chi^h\sum_{i<k}V_i^k}{M}+h\cdot C_k-1$$

$$\chi^h\sum_{i<k}V_i^k+M\cdot h\cdot C_k-M<M\cdot\chi^h \tag{3.30}$$

由(3.28)还可知$\chi^h<\left(h-1\right)T_k+D_k$，进而可知：

$$1+\left(\chi^h-D_k\right)/T_k<h$$

应用其不等式(3.30)可得：

$$\chi^h\sum_{i<k}V_i^k+M\cdot\left(1+\left(\chi^h-D_k\right)/T_k\right)\cdot C_k-M<M\cdot\chi^h$$

整理此不等式，可得：

$$\chi^h\left(\sum_{i<k}V_i^k+M\cdot U_k-M\right)<M\left(1+D_k\cdot U_k-C_k\right)$$

$$\chi^h<\frac{M\left(1+D_k\cdot U_k-C_k\right)}{\sum_{i<k}V_i^k+M\cdot U_k-M}$$

即存在一个 χ^h 的上限，这与h是任意选取且 $\left(\chi^h\right)_{h<1}$ 是一个严格递增的序列相矛盾，因此假设不成立，于是证得两个终止条件中必有一个被满足。　□

定理3.3说明对任意满足下述条件的任务集：

$$\sum_{i<k}V_i^k+M\cdot U_k\neq M$$

如果终止条件 $Term(h,k)$ 不被满足，则提出的响应时间分析过程将最终发现一个截止期错失。因此，尽管本文的任务模型为每一个任务指定了一个明确的相对截止期以使上述截止期错失可以被发现，但是整个响应分析过程的终止性并不依赖于一个特定的相对截止期值。也就是说，如果对任意 h 终止条件 $Term(h,k)$ 都不能被满足，那么在本章提出的分析中将不存在一个响应时间上限。最后仅剩的一种情况是：

$$\sum_{i<k}V_i^k+M\cdot U_k=M$$

在这种情况下，对某些任务集本章的响应时间分析过程会终止，而对某些任务集则不能。因为这是一种非常特殊情况，而且可以在响应时间分析过程开始前进行检查，并调整任务集参数使其变成其他两种情况中的一种，所以该情况并不会造成实际问题。

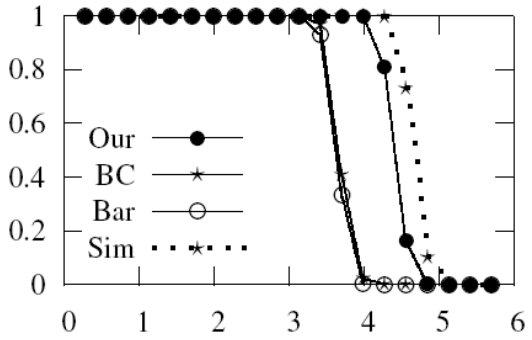
3.4　质量评价

本节通过考查接受率来评估所提出响应时间分析方法的质量。接受率为实验中（在某一特定设置下）可以被某特定可调度分析条件判断为可调度的任务集数与所有参与实验的任务集总数的比率。实验采用第 2.3.3 节中介绍的方法生产测试任务集。

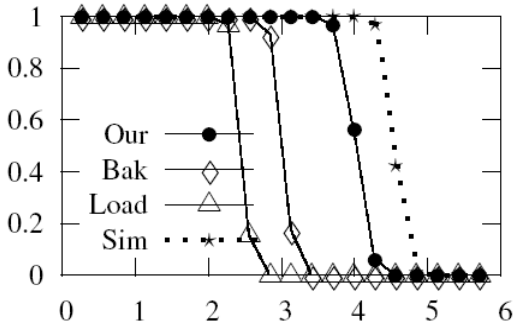
实验的默认设置如下：任务的优先级根据截止期单调算法分配，即具有最小相对截止期的任务优先级最高；处理器的个数为 6；对于每个任务 τ_i，其周期 T_i 均匀分布在区间 $[10,30]$ 内。在每组实验中，调整任务利用率 U_i 和相对截止期与周期之比 D_i/T_i 来生成不同特性的任务集。

图 3.7 是实验各组实验结果。在每个图表中，X 轴为资源利用率总和区间，Y 轴为接受率。图中曲线“Sim”表示模拟测试的接受率。模拟测试即通过模拟运行某任务集来判断其是否可调度。由于进行穷尽模拟（测试所有的任务释放便宜量和任务释放间隔）

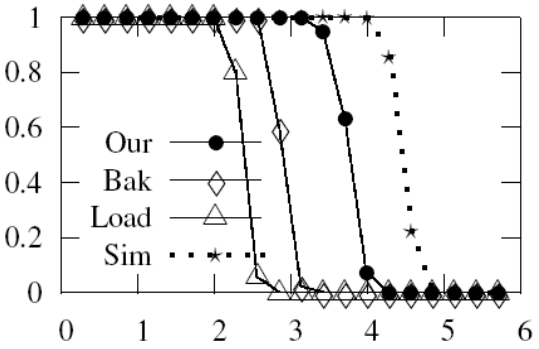
的计算复杂度太高而无法在有限时间内完成，本实验中将所有的任务偏移量都设置为0，使所有的任务都以严格周期进行释放。模拟实验将运行至该任务集中所有任务的公共周期（所有任务周期的最小公倍数）。通过这种方法得到的模拟测试结果可能会将一个原本不可调度的任务集判定为可调度。模拟测试可以作为所有任务集实际接受率的上限。



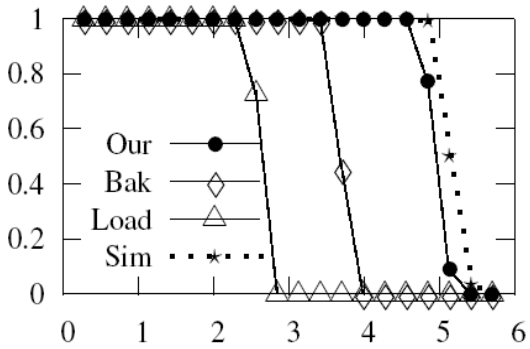
(a) $U_i \in [0.05, 0.1], \frac{D_i}{T_i} \in [0.8, 1]$



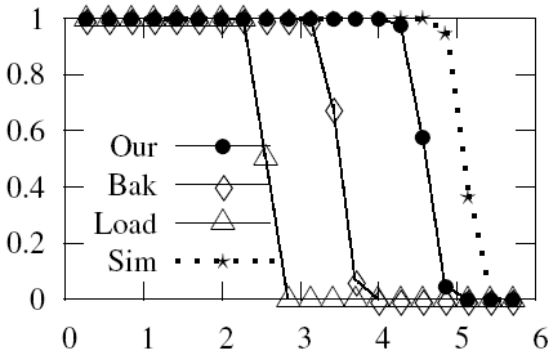
(b) $U_i \in [0.05, 0.15], \frac{D_i}{T_i} \in [0.8, 1]$



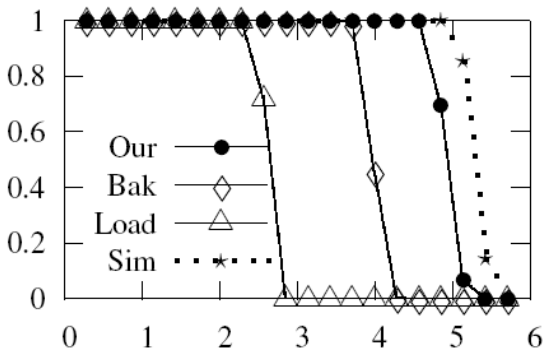
(c) $U_i \in [0.05, 0.2], \frac{D_i}{T_i} \in [0.8, 1]$



(d) $U_i \in [0.05, 0.1], \frac{D_i}{T_i} \in [0.8, 2]$



(e) $U_i \in [0.05, 0.2], \frac{D_i}{T_i} \in [0.8, 2]$



(f) $U_i \in [0.05, 0.2], \frac{D_i}{T_i} \in [0.8, 4]$

图 3.7 实验结果

Fig. 3.7 Experiment results

图 3.7-(a)(b)(c)对于限制截止期任务集来比较本章提出的响应时间分析方法（定理 3.1，图中记为“Our”）和已有方法的性能。在文献[41]中，比较了分析方法[BC-RTA]与其它一些已有的具有代表性的方法，并通过实验表明[BC-RTA]明显优于其他方法。因此，本文将不再阐述其它包括在[41]中的分析方法，而只和[BC-RTA]进行比较（在图中记为"BC"）。除此之外，实验还将和文献[41]中的可调度性分析方法进行比较，因为该方法没有被包括在[41]的比较中。实验结果表明在不同的任务利用率设置下，本章提出的响应时间分析方法都能够大幅度地提升任务集的接受率。其中对于任务利用率比较低的设置性能提升更加明显。

图 3.7-(d)(e)(f)显示了针对任意截止期任务，本章提出的响应时间分析方法与现有分析方法的性能比较。在图中，“Our”表示本章提出的分析方法（定理 2），“Bak”表示文献[121]中的可调度性分析方法，“Load”表示两个基于负载函数（Load Function）的可调度性分析方法[35,122]（如果一个任务集被这两个分析方法中的一个判定为可调度，则认为该任务集为可调度）。

实验结果表明，本章定理 3.2 中的分析方法得到的可调度性判定结果与模拟测试的结果非常接近。由于模拟测试的结果是系统实际接受率的近似上限，所以实际上本章定理 3.2 中的判定结果已经非常精确（尤其对于较低任务利用率的系统）。

实验还评估了本章定理 3.2 的运行效率，即可延展性（Scalability）。实验的参数如下：处理器个数为100；每个任务集中的任务个数均匀分布于区间[100,500]内；每个任务的周期均匀分布于区间[100,1000]内；每个任务的利用率均匀分布于区间[0.1,0.3]内；每个任务相对截止期 D_i 与周期 T_i 的比例均匀分布于区间[0.8,4]内。实验在一个 AMD Opteron 844（1.8GHz）服务器电脑上使用定理 3.2 的响应时间分析方法对1000 个这样的任务集进行判定，整个实验在三十分钟内完成，因此可以得出结论，本章定理 3.2 的分析方法平均可以在几秒内完成对一个实际大小任务系统的分析。

3.5　小结

本章介绍了一组新的全局固定优先级多处理机调度的响应时间分析方法。单处理机固定优先级调度中的响应时间分析技术在过去的二十多年里已经发展得非常成熟。但是，对于多处理机上全局调度响应时间分析的工作还相对较少。全局调度响应时间分析中的一个重要难题是没有像单处理机固定优先级调度中那样已知的关键时刻。因此，原有的全局调度响应时间分析技术需要对任务集的行为进行非常悲观的近似，从而获得安全的响应时间上限。本章提出的响应时间分析方法不但在理论上严格优于原有方法，而且在大部分情况下可以大幅度地改进了分析的精确性。该方法通过将[34]中的问题窗口扩

展技术应用于全局调度的响应时间分析，为全局固定优先级调度建立了一个与关键时刻类似的概念，称为近似关键时刻。该近似关键时刻表明任务的最大响应时间发生在所有高优先级任务，其中$M-1$个除外，同时释放任务实例的情况下。因此根据此近似关键时，对只需要对一小部分任务的工作量进行近似，因而大幅度地提高了分析的精确性。本章还将这一技术应用于任意截止期任务集。这是第一个对任意截止期任务集的全局固定优先级调度进行响应时间分析的工作。分析表明，上述的近似关键时刻同样适用于任意截止期任务集，因而也可以提供非常精确的分析。此外本章还研究了对于任意截止期任务集分析的终止性问题，即建立了全局固定优先级调度下具有有限响应时间的一般性条件。实验表明，本章提出的方法不但具有很高的分析精确度而且具有很好的执行效率。

第4章 不可抢占全局固定优先级调度分析

在实时调度研究领域，与抢占调度策略相比，不可抢占调度策略受到的关注相对较少。但是，在实际工业应用中，不可抢占调度策略通常比抢占调度策略的应用更加广泛。其原因有以下几个主要方面[123]：不可抢占调度比抢占调度更容易实现，且运行时的开销更低；在抢占调度中，由于缓存和流水线等硬件结构，对抢占造成的开销往往非常难以估计。不可抢占调度的优势在多核处理器平台上变得更加明显：在多核处理器上，由于任务迁移所引起的开销更高，且更加难以预测。但是，这些问题在不可抢占调度策略中则容易处理得多，因为每个任务实例都会一直执行到结束而任务迁移只发生在任务实例边界上。

不可抢占调度之所以在研究领域中被认为不适合实时系统，主要是由于其较差的响应能力。在单处理机系统中，一个高优先级任务可能由于无法抢占低优先级任务而被阻塞很长时间而错失其截止期。然而，这个问题在多核处理器上将得到很大缓解，因为多核处理器固有的并行性可以一定程度上抵消不可抢占阻塞带来的害处：在运行时，即使已经有若干个处理器被执行时间很长的低优先级任务占据，高优先级任务依然有机会在其它处理器上运行而满足其截止期约束。本章通过模拟实验来比较可抢占全局固定优先级调度算法（P-FP）和不可抢占全局固定优先级调度算法（NP-FP）的接受率。令人吃惊的是，在许多参数设置下，NP-FP 都比 P-FP 性能好（关于此实验的详细内容在第 4.5 节中介绍）。在上述实验中，并没有考虑抢占所带来的开销。而抢占开销将会进一步降低可抢占调度算法的性能。该实验结果表明，在多核处理器平台上对于许多实时系统不可抢占都可能是比可抢占调度更好的选择。

本章将主要研究多核处理器平台上周期性任务在 NP-FP 下的可调度性分析问题。所提出的分析技术是基于与上一章类似的问题窗口分析方法[28]和 Baruah 的问题窗口扩展技术[34]。本章提出的分析方法采用了上一章中的技术来限制前部任务个数从而获得较高的精确性。本章首先介绍一个通用的具有线性复杂度的可调度性分析判定条件，该判定条件适用于任何非主动空闲的多处理机调度算法。然后通过加强对问题窗口内任务负载的分析，提出一个针对 NP-FP 的具有二次方复杂度的可调度性判定方法。

4.1　相关工作

对于单处理机系统，Jeffay 等人研究了隐式截止期的周期任务系统的不可抢占调度问题，并提出了一个伪多项式复杂度的单处理机不可抢占 EDF 可调度性的精确判定方

法[123]。George 等人研究了任意截止期周期性任务的不可抢占调度问题，并提出了伪多项式复杂度的单处理机不可抢占 EDF 和不可抢占性固定优先级调度可调度性的精确判定方法[124]。Baruah 和 Chakraborty 研究了更一般性的任务模型的不可抢占调度问题，并证明了不可抢占性调度问题的分析可以被转化成为多项式数目个可抢占调度的分析问题[125]。

对于多处理机系统，Baruah 提出了一个全局不可抢占 EDF 的充分非必要可调度性判定条件[126]，称为[TEST-BAR]。[TEST-BAR]使用了与[87]中对可抢占 EDF 的分析相似的技术，但是引入了对由不可抢占阻塞造成的额外干涉的考虑。根据[TEST-BAR]，一个任务集如果满足下述条件则一定是 NP-EDF 可调度的：

$$V_{sum}(\tau) \le m-(m-1)V_{\max}(\tau) \tag{4.1}$$

其中

$$V_{sum}(\tau)=\sum_{\tau_i\in\tau}V_i,\quad V_{\max}(\tau)=\max_{\tau_i\in\tau}V_i$$

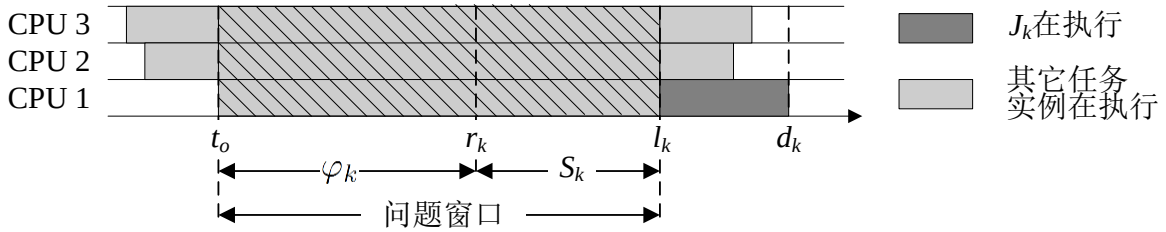
$$V_i=\begin{cases}\dfrac{C_i}{D_i-C_{\max}} & D_i>C_{\max}\\ \infty & D_i\le C_{\max}\end{cases}$$

$C_{\max}$ 为所有任务中的最长执行时间。显然，如果一个任务集满足 $C_{\max}\ge D_{\min}$，其中 $D_{\min}$ 所有任务中最小的相对截止期，则无论该任务集的资源利用率总和有多低其一定不能够通过上述可调度性判定条件。直观上讲，这意味着对于任何任务实例 J_k，如果存在某个任务具有能够覆盖其相对截止期的执行时间，则 J_k 一定不可调度。这种情况对单处理机调度一定是正确的，但是未必适用于多处理机调度。因为在多处理机系统中，即使某些处理器被具有很长执行时间的任务占据了，其它任务依然有机会在其它处理器上执行并满足截止期。最近，Baruah 的问题窗口扩展技术[34]被应用于 NP-EDF 的分析[127]，而本章将研究 NP-FP 的分析。此外本章提出的分析方法具有多项式时间复杂度（分别为线性和二次方时间复杂度），且适用于连续时间模型，而之前应用问题窗口扩展技术的分析[34, 127]都为伪多项式复杂度且只适用于离散时间模型。

4.2　一般性可调度性判定条件

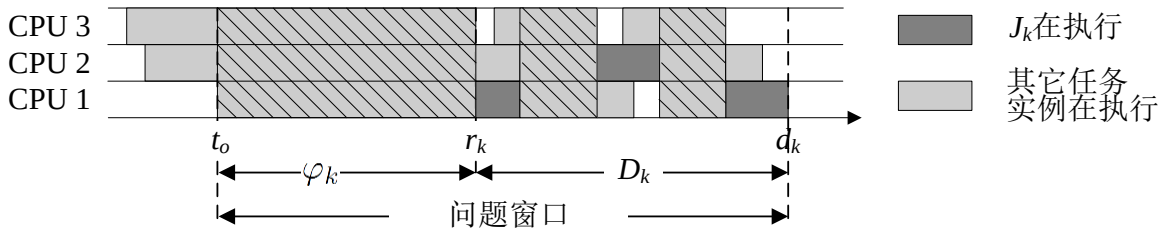
本节将介绍一个适用于任意非主动空闲不可抢占调度算法的一般性可调度性判定条件。给定一个非主动空闲不可抢占调度算法 *Schd*，假设一个任务集 τ 不能被 *Schd* 调度，令 J_k 为第一个错失截止期的任务实例，r_k 为 J_k 的释放时间。令 t_o 为 r_k 之前满足下列条件的最早的时间点：在时间区域 $[t_o, r_k]$ 内的任意时间点上都有 M 个任务在执行。令

$\varphi_k = r_k - t_o$，如图 4.1 所示。如果根据上述定义不存在这样一个 t_o，则令 $t_o = r_k$。



(a) 不可抢占调度中问题窗口

(a) Problem window in non-preemptive scheduling



(b) 不可抢占调度中问题窗口

(b) Problem window in preemptive scheduling

图 4.1 可抢占与不可抢占调度中问题窗口示意图

Fig. 4.1 Illustration of problem windows in preemptive and non-preemptive scheduling

根据 t_o 的定义，可知所有的处理器在时间区域 $[t_o, r_k]$ 都一直忙碌。因为调度算法不允许抢占，只要一个任务实例开始执行，则其必须执行至结束。因此，如果任务实例 J_k 在其最晚合理启动时间 l_k 之前开始执行，则它一定能够满足截止期。因为调度算法 *Schd* 是非主动空闲的，可知如果要使 J_k 错失截止期，，所有的处理器在时间区域 $[t_o, l_k]$ 内都必须一直忙碌。l_k 之后发生的事情对 J_k 的调度性没有影响。本章将时间区域 $[t_o, l_k]$ 称为问题窗口，如图 4.1(a)所示。

上述问题窗口的定义与第 3 章中的定义有相似之处，但又有区别。第 3 章的问题窗口中，所有处理器需要在时间区域 $[t_o, r_k]$ 内一直忙碌，但并不需要在时间区域 $[r_k, d_k]$ 中一直忙碌（只要其所有忙碌的时间段的总和足以使 J_k 错失截止期就可以，如图 4.1(b)）。

因此可知，使 J_k 能够错失截止期的一个必要条件是，任务集所有任务实例（除了 J_k ）在问题窗口内的工作量之和不少于 $(\varphi_k + S_k) \times M$ （图 4.1(a)中的阴影部分）。与第 3 章类似，为了求得任务集在问题窗口内的工作量总和，依然将每个任务在问题窗口内的工作量分为三部分：（1）前部任务实例的工作量；（2）中部任务实例的工作量；（3）后部任务实例的工作量。且有如下引理：

引理 4.1：最多 $M-1$ 个任务有前部任务实例。

使用 W 来表示任务集 τ 在问题窗口中的工作量总和，则可以为 W 找到一个上限，如

以下引理所述：

引理 4.2：用C_{M-1}^{sum}表示任务集τ中执行时间最短的$M-1$个任务的执行时间之和，则有如下W的上限值：

$$W \le C_{M-1}^{sum} + \sum_{\tau_i \in \tau} \left\lfloor \frac{\varphi_k + S_k}{T_i} \right\rfloor C_i + \sum_{\tau_i \in \tau} C_i \tag{4.2}$$

证明：对于任务集τ中任意一个任务τ_i，其在问题窗口内的工作量分为三部分：（1）前部任务实例的工作量，（2）中部任务实例的工作量，（3）后部任务实例的工作量。其中，前部任务实例和后部任务实例的工作量各为C_i。中部任务实例的个数最多为$\left\lfloor \frac{\varphi_k + S_k}{T_i} \right\rfloor$。因此，(4.2)式右部中的第二和第三项分别为所有任务的中部任务实例和后部任务实例工作量总和的上限。根据引理 4.1 可知最多$M-1$个任务有前部任务实例，因此，(4.2)式右部中的第一项为所有任务的前部任务实例工作量总和的上限。　□

本节可调度性判定条件的主要思想是，如果通过保证上述引理中任务集工作量总和的上限小于问题窗口中所有处理器提供处理能力的总和，即：

$$C_{M-1}^{sum} + \sum_{\tau_i \in \tau} \left\lfloor \frac{\varphi_k + S_k}{T_i} \right\rfloor C_i + \sum_{\tau_i \in \tau} C_i < (\varphi_k + S_k) M$$

说明J_k一定可以在问题窗口中开始执行（即在l_k之前开始执行）。这与J_k错失截止期的假设相矛盾，也就是说J_k一定能够满足截止期。通过将上述过程应用于τ中的每一个任务，就得到了一个判断τ可调度性的充分条件。值得注意的是，上述不等式中存在未知变量φ_k。可以使用如下事实来消掉这个变量：当φ_k增加时，一个任务在问题窗口中工作量中前部任务实例和后部任务实例所占的比例趋于下降，这意味着，由前部任务实例和后部任务实例引起的对任务工作量的多余估计对较小的φ_k值更加敏感。因此，可以通过将φ_k设为0来获得所期望的一般性可调度分析条件，如下定理所示：

定理 4.1（[TEST-1]）：用一个非主动空闲不可抢占多处理机调度算法*Schd*在 M 个处理器上来调度一个任务集τ。τ是可调度的，如果其满足下述条件

$$U(\tau) < M - \frac{\sum_{\tau_i \in \tau} C_i + C_{M-1}^{sum}}{S_{\min}} \tag{4.3}$$

其中C_{M-1}^{sum}表示任务集τ中执行时间最短的 M-1 个任务的执行时间之和，$S_{\min}$为所有任务松弛时间S_i的最小值。

证明：用反证法证明。假设一个任务集τ满足不等式(4.3)但是不能被*Schd*调度，且J_k是其中第一个错失截止期的任务实例。于是可知任务集在问题窗口内的工作量总和 W 一定不小于$(\varphi_k + S_k) \times M$，又根据引理 4.2 可得

$$C_{M-1}^{sum}+\sum_{\tau_i\in\tau}\left\lfloor\frac{\varphi_k+S_k}{T_i}\right\rfloor C_i+\sum_{\tau_i\in\tau}C_i\geq(\varphi_k+S_k)M$$

因为$\left\lfloor\frac{\varphi_k+S_k}{T_i}\right\rfloor\leq(\varphi_k+S_k)\frac{C_i}{T_i}$和$U(\tau)=\sum_{\tau_i\in\tau}\frac{C_i}{T_i}$可得

$$C_{M-1}^{sum}+(\varphi_k+S_k)U(\tau)+\sum_{\tau_i\in\tau}C_i\geq(\varphi_k+S_k)M\Rightarrow\sum_{\tau_i\in\tau}C_i+C_{M-1}^{sum}$$

$$\geq(\varphi_k+S_k)\left(M-U(\tau)\right)$$

又因为$\varphi_k\geq 0$且$S_k\geq S_{\min}$可得：

$$U(\tau)\geq M-\frac{\sum_{\tau_i\in\tau}C_i+C_{M-1}^{sum}}{S_{\min}}$$

这与τ满足不等式(4.3)的假设相矛盾。　□

可调度性判定条件[TEST-1]适用于任意非主动空闲不可抢占多处理机调度算法：上述证明除了要求 *Schd* 为非主动空闲和不可抢占以外对调度算法没有任何特殊要求。值得注意的是，可调度性判定条件[TEST-1]不像 TEST-BAR 那样受到下述限制：任意满足$C_{\max}\geq D_{\min}$的任务集都将被判定为不可调度。

$U(\tau)$，$\sum_{\tau_i\in\tau}C_i$和$S_{\min}$都可以在线性时间内计算出来，此外还可以通过使用线性时间选择算法[120]来计算C_{M-1}^{sum}，因此[TEST-1]具有线性时间计算复杂度。

4.3　对 NP-FP 改进的可调度性判定条件

上一节中介绍了一个适用于任意非主动空闲不可抢占多处理机调度算法的一般性可调度性判定条件[TEST-1]。[TEST-1]是安全的，但是却比较悲观，主要原因是其对任务集在问题窗口里的工作量的计算非常不精确。本节将介绍一个针对 NP-FP 的可调度性判定条件[TEST-2]，[TEST-2]将对问题窗口的定义以及任务集在问题窗口里工作量的计算进行改进，来获得更加精确的判定。

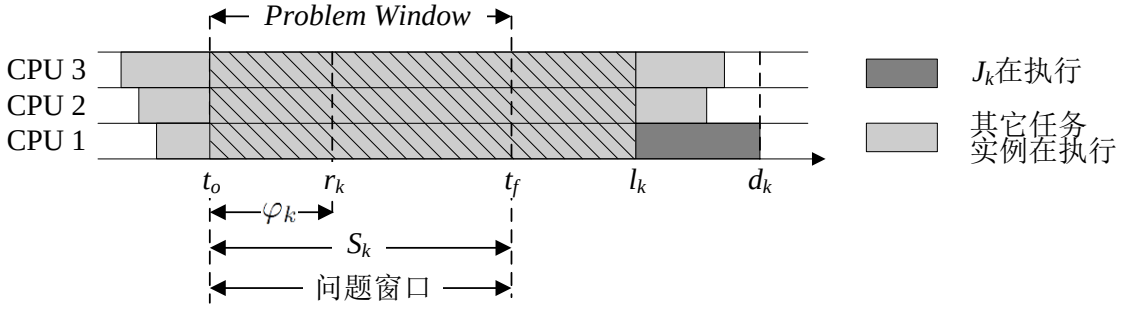


图 4.2 新定义的问题窗口示意图

Fig. 4.2 Illustration of the newly defined problem window

与上一节相似，假设一个任务集τ不能被 NP-FP 调度，令J_k为第一个错失截止期的任务实例，r_k为J_k的释放时间。下面，将定义一个新的问题窗口。令t_o为r_k之前满足下列条件的最早的时间点：时间区域$[t_o, r_k]$内的任意时刻t都满足下述两个条件之一：

1. $|\Theta(t,\tau_k)|=M$且$\Theta(t,\tau_k)$中所有的任务都在t时刻执行

2. $\Theta(t,\tau_k)$中的某些任务在t时刻没有执行

其中$\Theta(t,\tau_k)$为t时刻所有比τ_k优先级高的未完成任务的集合。如果不存在满足上述条件的t_o，则令$t_o=r_k$。令$t_f=t_o+S_k$，且定义时间区域$[t_o,t_f]$为问题窗口，如图 4.2 所示。根据该定义，尽管问题窗口的起始点t_o是未知的，但问题窗口的长度为固定值S_k。

根据这个新的问题窗口定义可知如下引理：

引理 4.3：所有在问题窗口$[t_o,t_f]$中执行的任务实例，要么比J_k优先级高，要么在t_o之前已经开始执行。

证明：证明将分为两步：首先第一步将证明在问题窗口$[t_o,t_f]$中所有的处理器都是一直忙碌的，然后利用第一步的结论第二步将证明任意在$[t_o,t_f]$内执行的任务实例要么比J_k优先级高，要么在t_o之前已经开始执行。

首先证明第一步。考虑t_o定义中的两个条件，对于任意时间点$t\in[t_o,r_k]$，如果第一个条件成立，则显然所有的处理器都忙碌；如果第二个条件成立，即存在某些未完成的任务不能执行，且 NP-FP 为非主动空闲调度策略，因此可知所有处理器在t时刻都为忙碌。因此，可知在时间区域$[t_o,r_k]$内所有的处理器都持续忙碌。根据t_f的定义可知$t_f\le l_k$，因此可知所有的处理器在时间区域$[r_k,t_f]$内持续忙碌。综上所述，可知所有的处理器在时间区域$[t_o,t_f]$内为持续忙碌。

下面用反证法证明第二步。假设一个优先级不高于J_k且在$t\in[t_o,t_f]$时刻开始执行的任务实例J_i在$[t_o,t_f]$内执行，考虑两种情况：（1）$t\in[t_o,r_k]$和（2）$t\in[r_k,t_f]$。

首先考虑情况（1）。因为J_i在t时刻开始执行，可知$\Theta(t,\tau_i)$中所有任务在t时刻都在执行且$|\Theta(t,\tau_i)|<M$。因为J_i的优先级不高于J_k，所以$\Theta(t,\tau_k)\subseteq\Theta(t,\tau_i)$。有此可知$\Theta(t,\tau_k)$中所有任务在$t$时刻都在执行且$|\Theta(t,\tau_k)|<M$，这与$t_o$的定义相矛盾。

下面考虑情况 2。因为J_k错失截止期，任何比J_k优先级低的任务实例都不可能在$[r_k,l_k]$内开始执行。此外，因为本章假设每个任务的相对截止期都不大于其周期，所以J_k的前继任务也不能在$[r_k,l_k]$内开始执行。又因为$t_f\le l_k$，因此可知任何优先级不高于J_k的任务实例都不能在$[r_k,t_f]$内开始执行，这与假设相矛盾。

综上所述，两种情况都将导致矛盾，因此假设不成立，引理得证。　□

引理 4.4：最多M个任务有前部任务实例。

证明：使用符号t_o^-来表示一个在t_o之前但是与t_o无限接近的时间点。因此，根据t_o的

定义可知，$|\Theta(t_o^-,\tau_k)|\neq M$ 且 $\Theta(t_o^-,\tau_k)$ 中的所有任务都在执行。因为不可能有多于 M 个任务同时执行，所以唯一可能的情况是 $|\Theta(t_o^-,\tau_k)|< M$ 且 $\Theta(t_o^-,\tau_k)$ 中的所有任务都在执行。一个优先级高于 J_k 的任务可能有前部任务实例的前提条件是它必须包含在 $\Theta(t_o^-,\tau_k)$ 中，因此至多有 $|\Theta(t_o^-,\tau_k)|$ 个优先级高于 J_k 的任务有前部任务实例。一个优先级不高于 J_k 的任务可能有前部任务实例的前提条件是它必须在 t_o^- 时刻执行。因为 $|\Theta(t_o^-,\tau_k)|$ 个处理器被比 J_k 优先级高的任务实例占用了，因此在 t_o^- 时刻执行的优先级不高于 J_k 的任务最多为 $M-|\Theta(t_o^-,\tau_k)|$ 个。因此，最多有 $|\Theta(t_o^-,\tau_k)|+(M-|\Theta(t_o^-,\tau_k)|)=M$ 个任务具有前部任务实例。　□

使用 $I_k^{NC}(\tau_i)$ 来表示一个任务 τ_i 没有前部任务实例时其在问题窗口内工作量的一个上限，使用 $I_k^{CI}(\tau_i)$ 来表示一个任务 τ_i 有前部任务实例时其在问题窗口内工作量的一个上限。下面将介绍如何计算 $I_k^{NC}(\tau_i)$ 与 $I_k^{CI}(\tau_i)$：

- 计算 $I_k^{NC}(\tau_i)$

– τ_i 优先级高于 τ_k。这种情况下可以通过下式计算 $I_k^{NC}(\tau_i)$（证明与第 3 章相似）：

$$I_k^{NC}(\tau_i)=\left\lfloor \frac{S_k}{T_i} \right\rfloor C_i + \left(S_k \bmod T_i \right)_0^{C_i}$$

– τ_i 优先级不高于 τ_k。根据引理 4.3 可知，对于任意优先级不高于 τ_k 的任务，只有其在 t_o 以前开始的任务实例可能在问题窗口里执行。因此此类任务没有前部任务实例，即

$$I_k^{NC}(\tau_i)=0$$

- 计算 $I_k^{CI}(\tau_i)$

– τ_i 优先级高于 τ_k。这种情况下可以通过下式计算 $I_k^{NC}(\tau_i)$（证明与第 3 章相似）：

$$I_k^{CI}(\tau_i)=\left\lfloor \left(S_k - C_i \right)_0 / T_i \right\rfloor C_i + C_i + \alpha^{S_k}$$

其中

$$\alpha=\left(S_k - C_i \right)_0 \bmod T_i - (T_i - D_i)_0^{C_i}$$

这里对 $I_k^{CI}(\tau_i)$ 加上了上限 S_k，这是因为一个任务在某个时间区域内的工作量不可能超过这个时间区域的长度。

– τ_i 优先级不高于 τ_k。根据引理 4.3 可知，对于任意优先级不高于 τ_k 的任务，只有其在 t_o 以前开始的任务实例可能在问题窗口里执行。因此此类任务最多有一个前部任务实例，即

$$I_k^{CI}(\tau_i) = C_i^{S_k}$$

上述计算中一个重要的性质是，$I_k^{NC}(\tau_i)$ 与 $I_k^{CI}(\tau_i)$ 都完全与 t_o 这个时间点在哪里无关。这是因为本节定义的问题窗口为一个长度固定为 S_k 的时间区域。定义 Ω_k 为所有任务在问题窗口中工作量的最大上限制，则通过第 3 章中的图 3.4 中的算法可以求得 Ω_k。

如果 Ω_k 比整个问题窗口内所有处理器所提供的计算能力小，则 τ_k 可以在问题窗口里开始执行，又因为问题窗口的结束时间 t_f 不晚于 l_k，可知 J_k 一定能满足截止期。通过将上述推理应用于任务集中的每个任务，就得到了一个 NP-FP 可调度性判定的充分条件。

定理 4.2：如果一个任务集中的每个任务 τ_k 都满足条件 $\Omega_k < S_k M$，则该任务集可被 NP-FP 调度。

4.4 判定条件的可持续性

定理 4.3：可调度性判定条件[TEST-1]关于执行时间，相对截止期和最小释放间隔都是自可持续的。

证明：直接通过观察判定条件[TEST-1]可得。　□

定理 4.4：可调度性判定条件[TEST-2]关于执行时间，相对截止期和释放间隔都是可持续的，且关于最小释放间隔为自可持续的。

证明：令 τ_k 为一个可以通过[TEST-2]判断条件的任务。[TEST-2]独立于 T_k，且 $I_k^{NC}(\tau_i)$ 与 $I_k^{CI}(\tau_i)$ 都是关于 T_i 非增单调的，因此如果增大某个任务的最小释放间隔[TEST-2]依旧成立。因此[TEST-2]关于最小释放间隔都是自可持续的，这也意味着它是最小释放间隔都是可持续的。

下面证明[TEST-2]是关于执行时间可持续的。仍然令 τ_k 为一个可以通过[TEST-2]判断条件的任务，这意味着 τ_k 的任意一个实例都可以在其最晚可行开始时间之前开始执行，因此减小 τ_k 的执行时间后这些任务实例仍然可以满足截止期。此外，$I_k^{NC}(\tau_i)$ 和 $I_k^{CI}(\tau_i)$ 都是关于 C_i 非增单调的，因此减小其它任务的执行时间也不会使 τ_k 承受更多的干涉而变得不可调度。

最后来证明[TEST-2]是关于相对截止期可持续的。在固定优先级调度中，调度行为与任务的相对截止期是无关的，因此在一个具体的任务释放序列中，如果改变某个任务的相对截止期，其调度行为不变，因此增大某个任务的相对截止期不会使某个原本可调度的任务变得不可调度。　□

[TEST-2]不是关于执行时间和相对截止期自可持续的。问题出现在当减小被分析任

务的执行时间或者相对截止期时：减小被分析任务的执行时间或者相对截止期可能会增大问题窗口的长度 S_k。由于 Ω_k 是关于 S_k 的分段线性函数，且增加 S_k 会使得[TEST-2]判定条件式的左部增长比其右部增长快。例如图 4.3 中的任务集在两个处理器上调度，其中任务 τ_5 为目前正在分析的任务（即为 τ_k）。在现有的参数设定下，可知 $S_5=10$ 且

$$\Omega_5=10+8+0.9+0.9=19.8<10\times 2=S_5\times M$$

其中 τ_1 和 τ_2 有前部任务实例，τ_3 和 τ_4 没有。根据上述计算可知，τ_5 是可调度的。但是，如果使 C_5 减少 1 或者 D_5 增加 1，则 $S_5=11$，且有

$$\Omega_5=11+8+1.8+1.8=22.6>11\times 2=S_5\times M$$

由上述结果可知[TEST-2]判定 τ_5 为不可调度的。由此可见，[TEST-2]不是关于[TEST-2]不是关于执行时间或者相对截止期自可持续的。但是，使除被分析任务以外的其它任务的执行时间变小或者相对截止期变大不会使被分析任务变得不可调度。因此，在系统设计过程，当一个任务的参数变为“更好”时，设计者只需要重新考查这个任务是否依旧能通过可调度性测试，而不需要担心其它任务。

任务	T_i	D_i	C_i
τ_1	10	10	6
τ_2	10	10	4
τ_3	10	10	0.9
τ_4	10	10	0.9
τ_5	14	14	4

图 4.3 一个说明[TEST-2]不是 Ck 和 Dk 关于自可持续的任务集例子

Fig. 4.3 A task example showing [TEST-2] is not self-sustainble with respect to Ck and Dk

4.5 质量评价

本节将使用随机生成的任务集来评估所提出的可调度性判定条件的接受率。图 4.4(a)实验中的任务参数如下：处理器的个数为 4，每个任务 τ_i 的周期 T_i 均匀分布于区间 $[10,20]$ 内；相对截止期 D_i 与周期（最小释放间隔）T_i 的的比例均匀分布于区间 $[0.9,1]$ 内，资源利用率 U_i 均匀分布于区间 $[0.1,0.4]$ 内。在图 4.4-(b)至 4.4-(f)的实验中改变关于 T_i 和 U_i 的参数而保持其它参数不变。实验结果表明，一般性可调度性判定条件[TEST-1]的接受率比较低，这是因为它计算任务集在问题窗口内工作量的方法过于悲观。相比之下，针对 NP-FP 的可调度性判定条件[TEST-2]的接受率要高出[TEST-1]很多。

下面将比较 NP-FP 和 F-FP 两种调度算法在模拟实验下的接受率。对于硬实时系统，通常认为一个调度算法的判定条件的接受率比算法本身的接受率更加有意义。但是，通

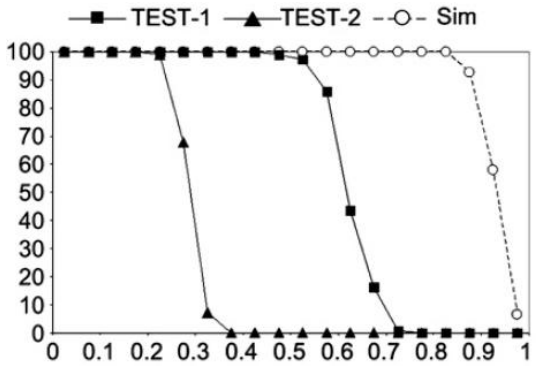
过研究一个算法本身的接受率对于发掘算法分析的“潜力”并以此作为开发新的分析方法的依据有重要意义。

因为不可能在有限时间内对全局多处理机调度的状态进行穷举，这里将使用仅覆盖部分状态空间的模拟。对于每个任务τ_i，其初始释放偏移量在区间$[0,T_i]$内随机选取；当一个任务τ_i从上次释放已经经历了其最小释放间隔T_i以后，每个时刻，它都有 3/4 的概率释放（也就是说，有 1/4 的概率不在当前时刻释放，而等待到下一时刻）。每个任务集的模拟起始于时间 0，终止于$\max(5\times10^6, hyperperiod)$。对每组参数设置，实验对$5\times10^5$组任务集进行模拟。

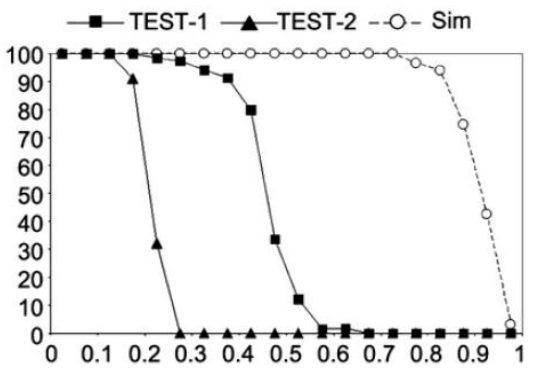
在图 4.5 的实验中，处理器的个数为 4，每个任务的资源利用率均匀分布于区间$[0.1,0.4]$。图 4.5(a)的实验表明当任务的周期范围变大时（保持资源利用率范围不变），任务的执行时间范围也变大，因此会造成较大的不可抢占阻塞，这对 NP-FP 的可调度性不利。但是，图 4.5(b)的实验表明改变周期范围对 P-FP 的影响较小。

在图 4.6 的实验中，处理器的个数为 8，每个任务的周期均匀分布于区间$[10,80]$。图 4.6(a)的实验表明当任务的资源利用率变大时，任务的执行时间范围也趋于变大，因此会造成较大的不可抢占阻塞，这对 NP-FP 的可调度性不利。但是，图 4.6(b)的实验表明，增大任务资源利用率对 P-FP 的影响较小。

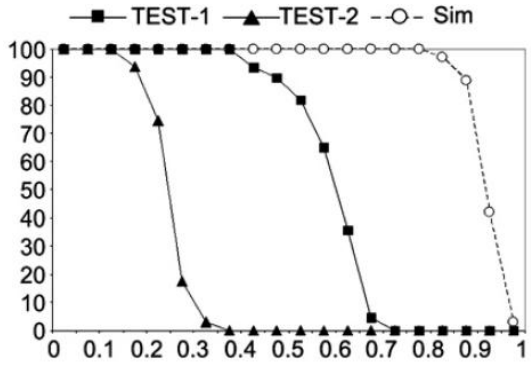
在图 4.7 的实验中使用不同的处理器数目，而保持每个任务的周期与资源利用率不变（$T_i\in[10,320]$，$U_i\in[0.1,0.4]$）。图 4.7-(a)中的实验表明当 NP-FP 的性能随着处理器数目的增加而变得更好。尽管本组实验的任务周期范围很大，因此造成的不可抢占阻塞也很大，但是 NP-FP 的性能可以通过增加处理个数得到补偿。这是一个非常重要的现象：因为多核芯片上处理核心的数目在迅速增长，在未来众核系统中不可抢占阻塞所带来的影响将越来越小。相比较之下，图 4.7(b)的实验表明增大处理器个数对 P-FP 影响较小。



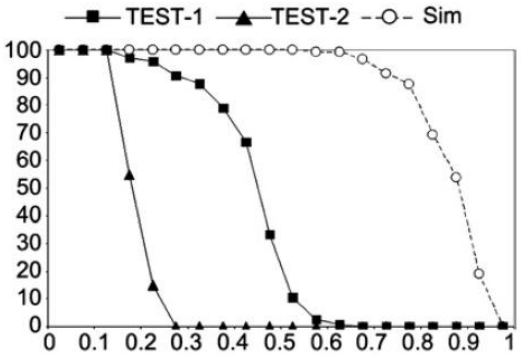
(a) $U_i\in[0.1,0.4], T_i\in[10,20]$



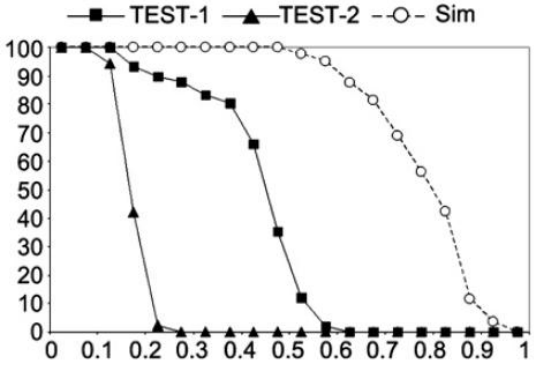
(b) $U_i\in[0.1,0.6], T_i\in[10,20]$



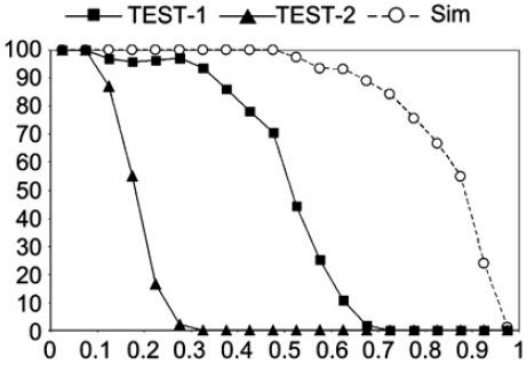
(c) $U_i \in [0.1, 0.4], T_i \in [10, 40]$



(d) $U_i \in [0.1, 0.6], T_i \in [10, 40]$



(e) $U_i \in [0.1, 0.4], T_i \in [10, 60]$

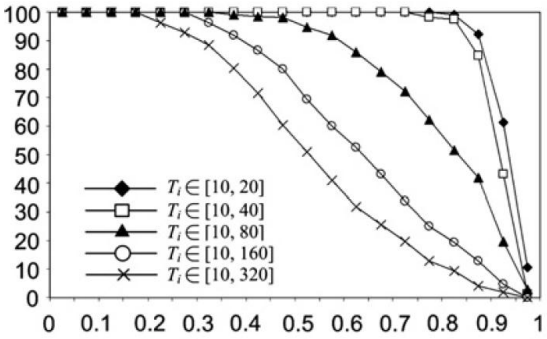


(f) $U_i \in [0.1, 0.6], T_i \in [10, 60]$

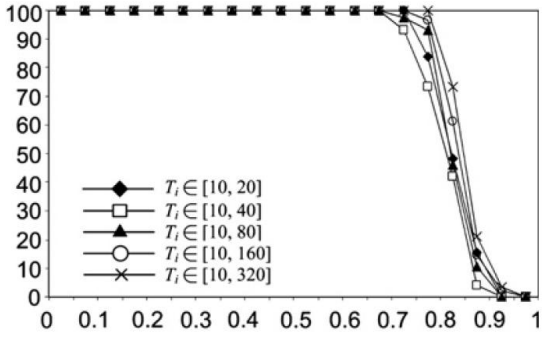
图 4.4 接受率实验结果

Fig. 4.4 Experiment results of acceptance ratio

上面实验中的另一个重要现象是，在许多参数设置下，NP-FP 的性能都优于 P-FP（比如图 4.5 中 $T_i \in [10, 20]$ 和 $T_i \in [10, 40]$ 的实验，图 4.6 中 $U_i \in [0.1, 0.2]$ 的实验以及图 4.7 中 $M = 64$ 的实验）。一般认为，不可抢占调度的实时性能要低于抢占性能。而上述现象与此相矛盾（需要注意的是，上述实验并没有考虑上下文切换所带来的影响)。下面直观地解释出现这种现象的原因。



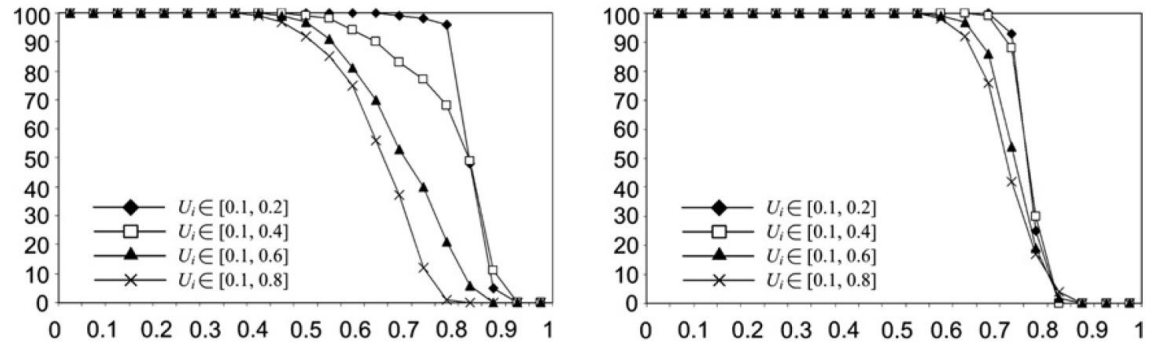
(a) NP-FP



(b) P-FP

图 4.5 不同周期范围下的模拟实验结果 (M=4, Ui∈[0.1, 0.4])

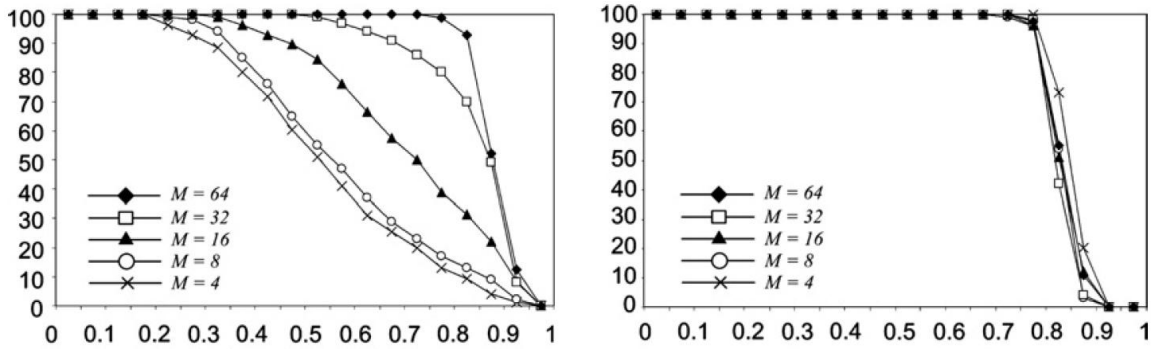
Fig. 4.5 Experiment results of simulation with different period ranges (M=4, Ui∈[0.1, 0.4])



(a) NP-FP　　(b) P-FP

图 4.6 不同任务资源利用率范围下的模拟实验结果 (M=8, Ti∈[10, 80])

Fig. 4.6 Experiment results of simulation with different utilization ranges (M=8, Ti∈[10, 80])



(a) NP-FP　　(b) P-FP

图 4.7 不同处理器个数下模拟实验结果 (Ti∈[10, 320], Ui∈[0.1, 0.4])

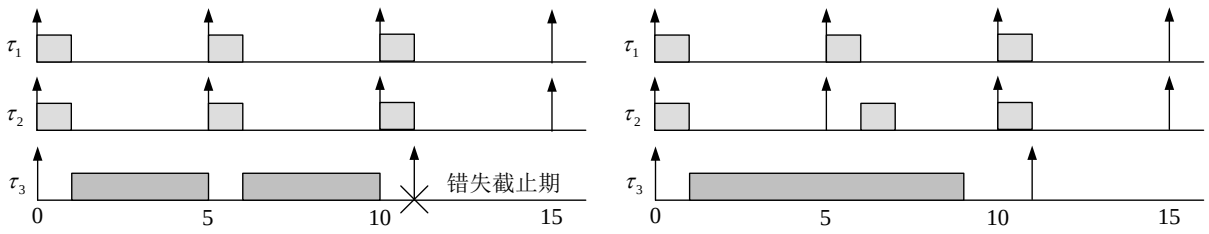
Fig. 4.7 Experiment results of simulation with different processor numbers (Ti∈[10, 320], Ui∈[0.1, 0.4])

在不可抢占调度中，一个任务实例一旦开始，就将执行至结束，这有利于低优先级任务的可调度性。但是与此同时，低优先级任务会造成对高优先级任务的不可抢占阻塞，这对高优先级任务的可调度性是有害的。在单处理机平台上，不可抢占阻塞带来的害处是占统治地位的，因为一旦存在一个执行时间很长的低优先级任务，则所有相对截止期较短的高优先级任务就一定会错失截止期。但是，在多处理机上情况有所不同：即使存在一个执行时间很长的低优先级任务占用某一个处理器，高优先级任务仍旧有机会在其它处理器上执行并满足截止期。如图 4.7(a)中所示，随着处理器数目的增加，不可抢占阻塞的影响就越小。因此，当不可抢占阻塞的影响足够小而低优先级任务因不可抢占而获得的收益占统治地位时，NP-FP 将呈现比 P-FP 更好地实时性能。

任务	T_i	D_i	C_i	优先级
τ_1	5	5	1	高
τ_2	5	5	1	中
τ_3	11	11	8+ε	低

图 4.8 一个可以被 NP-FP 调度却不能被 P-FP 的任务集例子

Fig. 4.8 A task set example that is schedulable by NP-FP but not P-FP



(a)　　(b)

图 4.9 上图中任务集分别在 NP-FP 和 P-FP 下的调度情况

Fig. 4.9 Schedule of the task set in Fig. 4.8 under NP-FP and P-FP

例如，考虑图 4.8 中的任务集运行在 2 个处理器上。图 4.9 所示为其分别在 P-FP 和 NP-FP 下的调度情况(假设每个任务的初始释放偏移量为 0 且为严格周期释放)。图 4.9(a)表明在 P-FP 下任务 τ_3 将错失其截止期，尽管有相当的处理器资源是空闲的。但是，如图 4.9(b)所示，在 NP-FP 下 τ_3 将满足其截止期。同时，由 τ_3 引起的不可抢占阻塞又不足以使任务 τ_1 和 τ_2 措施截止期。实际上该任务集可以通过判定条件[TEST-2]，因此即使没有图 9 中对其释放时间的假设它依旧是可调度的。

综上所述，NP-FP 的性能严重依赖于任务系统的参数特点，而 P-FP 的性能则在不同的参数设置下差别不大。总的来说，在下述情况中 NP-FP 的实时性能优于 P-FP：（1）任务周期的范围比较狭窄;（2）任务的资源利用率比较低;（3）处理器的数目比较大。因为多核芯片上所集成的处理器的数目在迅速增加，在未来的众核系统中，不可抢占调度将具有更大的优势。

4.6　小结

本章中研究了不可抢占全局固定优先级调度算法（NP-FP）的可调度性分析问题。通过使用随机生成任务集进行大量的模拟实验发现了一个意外的现象：在很多参数设置之下，不可抢占全局固定优先级调度算法（NP-FP）比可抢占全局固定优先级调度算法（P-FP）的实时性能要好，这与传统对关于 P-FP 的实时性能总是优于 NP-FP 的认识相矛盾。这也意味着，对于许多基于多核处理器的实时应用，不可抢占调度可能是一个比 P-FP 更好地选择（尤其如果同时考虑 NP-FP 在实现复杂度，运行时开销等其它方面的优势)。为了深入地探索不可抢占全局固定优先级调度算法的本质，本章提出了两个可调度性判定条件。首先介绍一个通用的具有 $O(N)$ 复杂度的可调度性分析判定条件，该判定条件适用于任何非主动空闲的多处理机调度算法。然后通过加强对问题窗口内任务负载的分析，来提出一个针对 NP-FP 的具有 $O(N^2)$ 复杂度的可调度性判定方法。

第5章　一种全局固定实例优先级调度算法及分析

如第 1、2 章中所述，将单处理机调度中最优调度算法 RMS 和 EDF 直接应用于全局调度可能导致非常低的实时性能，即 Dhall 效应。但是以往的许多研究工作依然考虑使用基于 RMS 和 EDF 的全局调度。其主要原因之一是，研究者们期望全局 RMS 和 EDF 调度的 Dhall 效应只是在极端特殊的情况下才会发生，而对于普通任务集仍然有较好的质量。Davis 和 Burns 通过研究全局固定任务优先级调度中的优先级分配问题发现，对于全局固定优先级调度来说任务的优先级顺序对调度性能起着关键作用，而且在大多数情况下根据 RMS 进行优先级分配会导致较差的性能。这也引发了对另一个重要问题的思考：EDF 优先级分配策略在全局多处理机调度中依然适用吗？

本章提出了一个新的全局固定实例优先级调度算法 JPA（Job-level Priority Assignment，实例级别优先级分配）。该算法可以看作是固定优先级调度和 EDF 调度的推广：JPA 可以通过让一个任务所有实例的优先级都高于另一个任务的所有实例来模拟固定优先级调度；如果任务实例的优先级严格按照它们的绝对截止期顺序分配，则 JPA 退化成 EDF。JPA 进行实例基本的优先级分配的灵活性使其有很大机会成功调度那些在固定优先级或 EDF 下不可调度的任务集。

进行实例级别的优先级分配中的一个重大挑战是任务实例的运行时信息是未知的：一个任务实例的释放时间和绝对截止期只有其在运行时被释放以后才可知。为了解决这个问题，JPA 将不会直接对每个运行时任务实例进行优先级分配，而是在设计阶段先为有限个任务实例进行抽象的优先级分配，然后在运行时使用这个信息结合系统的运行状态来决定每个任务实例的优先级。

JPA 在活动更强的调度能力的同时，带来了比较大的运行时开销（需要动态维护一些较大的数据结构）。一些对开销敏感的嵌入式系统可能更希望使用更加高效的调度算法。因此，本章在 JPA 的基础之上，又提出了一种改进的算法 JPAZ，这个改进的算法之需要维护很小的运行时数据结构，但依然保持非常高的调度能力。

5.1　实例级别优先级分配调度算法 JPA

至少有两个问题使对周期性任务系统进行灵活的实例级别优先级分配非常困难：（1）一个周期性任务系统在运行时会释放无穷多个任务实例，因此在有限的设计时间内无法对无穷多个任务实例一一进行优先级分配（并保证它们的可调度性）；（2）任务实例的信息是不完整的，因为一个任务实例的释放时间和绝对截止期只有在其运行时真

正释放以后才可知。

为了解决上述问题，JPA 将不会直接对每个运行时任务实例进行优先级分配，而是在设计阶段先为有限个任务实例进行抽象的优先级分配，然后再运行时使用这个信息结合系统的运行状态来决定每个任务实例的优先级。更详细地讲，JPA 分为两个部分：设计阶段的优先级分配和运行时调度。在设计阶段，JPA 为一个最大忙碌期内所可能释放的所有任务进行优先级分配，然后这些信息将用于 JPA 运行时的调度。每个任务的可调度性仅取决于设计阶段的优先级分配，也就是说，如果 JPA 能为最大忙碌期内的任务实例成功地找到一个优先级分配方案，那么根据这个信息，JPA 的运行时调度将保证整个系统在运行时不会发生截止期错失。

5.1.1 设计阶段优先级分配

忙碌期是一个所有处理器始终忙碌的连续时间区域。对于任意正规化资源利用率总和小于 1 的任务系统，可以为忙碌期的最大长度找到一个上限UB。假设任意一个长度为L的时间区域$[t_1,t_2]$。一个任务τ_i在这个时间区域内能执行的任务实例的个数不超过$\lfloor L/T_i \rfloor+2$，因此该任务集在这个时间区域内的工作量总和一定不超过

$$\sum_{\tau_i\in\tau}(\lfloor L/T_i \rfloor+2)\times C_i \le U(\tau)\times L+2\sum_{\tau_i\in\tau}C_i$$

另一方面，如果时间区域$[t_1,t_2]$是忙碌期，即$[t_1,t_2]$内没有处理器空闲，则该任务集在这个时间区域内的工作量总和至少应该为$L\times M$。因此可知

$$U(\tau)\times L+2\sum_{\tau_i\in\tau}C_i \ge L\times M$$

通过求解上述不等式可得

$$L \le \frac{2\sum_{\tau_i\in\tau}C_i}{M-U(\tau)}$$

即如果任务集的正规化资源利用率总和小于 1，L一定小于一个有限值。

JPA 的设计阶段优先级分配只考虑能够在一个忙碌期内释放的任务实例。每个任务τ_i在忙碌期内释放的任务实例的个数为$n_i=\lceil \mathrm{UB}/T_i \rceil$，且所有任务释放忙碌期内释放的任务实例的总和为$n=\sum_{\tau_i\in\tau}n_i$。JPA 设计阶段优先级分配的目标是将$[1,n]$中的每一个优先级数值赋给一个任务实例，其中较小的数值代表较高的优先级。JPA 设计阶段优先级分配的结果是为每个任务τ_i生成一个优先级列表Ψ_i。Ψ_i中按优先级有高至低的顺序记

录了所有分配给τ_i的任务实例的优先级数值。

1：$\forall \tau_i \in \tau$：$\delta_i \leftarrow n_i$
2：**for** each ρ from n to 1 **do**
3：　**if** $\exists \tau_k$: TEST$\left(\tau_k, \{\delta_j\}_{\tau_j \in \tau}\right)$ = true **then**
4：　　Add ρ to Ψ_k
5：　　$\delta_k \leftarrow \delta_k - 1$
6：　**else**
7：　　**return** Failure
8：　**end if**
9：**end for**
10：**return** Success

图 5.1 JPA 的设计阶段优先级分配算法

Fig. 5.1 The offline priority assignment algorithm of JPA

优先级分配是使用所谓的“Audsley 方法”[128]进行的。这个方法自下而上地为n个任务构建一个优先级顺序，如图 5.1 中的算法所示。对每个任务τ_i，使用δ_i来表示其目前为止还未被分配优先级的任务实例的个数。在算法一开始，δ_i被初始化成n_i。然后算法从最低的优先级开始分配（n为最低优先级）。在分配的每一步，将找到一个满足下述测试条件的任务τ_k分配当前优先级数值ρ：

$$\text{TEST}\left(\tau_k, \{\delta_j\}_{\tau_j \in \tau}\right)$$

这个测试条件的功能是判断运行时τ_k释放的所有优先级为当前ρ的任务实例是否可以被调度。如果可以找到一个满足该测试条件的任务τ_k，则将当前ρ加入τ_k的优先级列表Ψ_k，然后将δ_k减去 1 并进入下一轮循环。如果有多个任务可以满足上述测试条件，则从中任意选取一个。如果在算法执行的某一轮找不过满足测试条件 TEST() 的任务，则优先级分配过程宣告失败，否则所有$[1,n]$的优先级数值都被加入到某个任务的优先级列表中去且优先级分配过程成功。

下面将首先介绍 JPA 的运行时调度。然后在第 5.3 节中将详细介绍测试条件 TEST()，并证明它确实能保证在运行时τ_k的所有优先级为当前ρ的任务实例可以被调度。由此也证明了整个优先级分配过程的成功可以保证系统的可调度性。

5.1.2 运行时调度

首先介绍 JPA 运行时调度的直观意义。任务τ_k一个实例J_k的运行时可调度性取决于在一个优先级在J_k之上的忙碌期（又称为J_k－忙碌期）内系统所产生的工作量。其中

J_k −忙碌期的定义为一个期间只有优先级不低于 J_k 的任务实例在执行的连续时间区域（空闲处理器被看作在执行一个最低优先级 $\perp$ 的空闲任务）。条件 TEST() 在对应于 J_k 优先级进行的测试相当于“模仿” J_k −忙碌期内可能发生的最大工作量，并以此来保证 J_k 的可调度性。

用 TEST() 来保证任务实例运行时刻调度性存在两个难点。首先，在运行时每个任务实例需要知道其相应的忙碌期是何时开始的，这样才能去使用对应任务的优先级列表中正确的优先级值。例如，如果一个任务实例 J_k 对应的忙碌期是从其释放时间开始的，那么这个任务实例的行为就对应设计阶段优先级分配中 J_k 在忙碌期内所释放的第一个任务实例，也就是说， J_k 应该使用优先级列表 Ψ_k 中的第一个优先级数值。这个问题是通过让 JPA 的运行时调度在每次抢占发生时来更新未释放任务与优先级列表中优先级数值的映射关系来解决的。其背后的机理是，每当一个低优先级被抢占时，都意味着比其优先级更高任务的一个新的忙碌期开始了。

第二个难点是，在运行时当一个忙碌期开始，可能还存在此前已经释放但是还未完成的任务实例。这本质上是和标准的全局固定优先级或 EDF 调度中的前部任务实例工作量相同的问题。这些前部任务实例需要被区别对待，以避免有其它不紧急任务所造成的不必要的干扰。JPA 通过将这些前部任务实例变得不可调度来解决这个问题。因此，在一个任务实例释放后所发生的任何优先级映射关系更新都不会对该实例造成干扰。这背后的主要设计思想是，这些变得不可抢占的任务实例不会对其它任务实例的可调度性分析造成影响。这是因为全局调度的关键时刻是未知的，因此在其分析中无论如何也要考虑前部任务实例工作量的影响。

JPA 的运行时调度可以概括为以下规则：

1. 每个任务实例在释放时被赋予一个优先级。这个优先级在该任务实例执行的整个期间都不会改变，但是一个任务实例在执行过程中可能会变成不可抢占的。

2. 任何时刻，一个任务的未释放实例和其优先级列表中的优先级数值之间存在一个映射关系。一个任务实例根据此映射关系在释放时获得优先级。

3. 每当发生抢占时，更新每个任务的未释放实例和其优先级列表中的优先级数值之间的映射关系（包括一个任务实例在一个空闲处理器开始执行的情况，因为这可以看作是抢占了空闲任务）。

下面详细介绍 JPA 的运行时调度。首先介绍索引列表 Λ_i 的概念。一个任务 τ_i 的索引列表 Λ_i 维护其未释放实例和其优先级列表中的优先级数值之间存在映射关系。 Λ_i 中含有若干索引，每个索引指向 τ_i 的优先级列表 Ψ_i 中的某个优先级值。 Λ_i 中的索引是按照它们所指向的优先级数值的顺序排列的。在 Λ_i 的初始状态中，其所含的索引与 Ψ_i 中的

优先级值是一一对应的。

当某个任务实例 J_i 释放时，优先级管理例程 $\mathrm{PrtManage}(J_i)$ 被调用，其伪代码如图 5.2 所示。其中的子例程 $\mathrm{RedirectToHigh}(S_k)$（第 11 行将）$S_k$ 中所有比指向 $\mathrm{prt}(J_L)$ 优先级高的索引都重定向到尽可能高的优先级（两个索引不能指向同一个优先级值）。需要注意的是，刚释放任务实例 J_i 所得到的优先级 $\mathrm{prt}(J_i)$ 一定不低于 p_{rls}。这是因为在 $\mathrm{RedirectToHigh}(S)$ 中每个索引要么不发生改变，要么被重定向到一个更高的优先级。此外，当一个任务实例 J_i 被释放时，其索引列表一定不为空。这是因为索引列表里的索引一定足够覆盖忙碌期内所有释放的任务，且每当一个新的忙碌期开始时（当一个任务释放且存在空闲处理器时）Λ_i 都被重置为初始状态。

```
1: if any processor is currently idle then
2:    Reset all tasks' index lists to the initial states
3:    All the running jobs become non-preemtable
4:else
5:    J_L ← lowest-priority preemptable running job
      (J_L ← NULL if all running jobs are non - preemptable)
6:    prls ← the priority pointed by the first index in Λ_i
7:    if J_L ≠ NULL and prls ▷ prt(J_L) then
8:       All running jobs except J_L become non-preemptable
9:       for each τ_k do
10:         S_K ← the set of indices in Λ_K pointing to priorities
                  higher than prt(J_L)
11:         RedirectToHigh
12:      end for
13:   end if
14:end if
15:prt(J_i) ← the priority pointed by the first index in Λ_i
16:Remove the first index in Λ_i
```

图 5.2 优先级管理例程 $\mathrm{PrtManage}(J_i)$

Fig. 5.2 The priority management routine $\mathrm{PrtManage}(J_i)$

在运行时的每一时刻可能会发生一些事件来改变系统状态。对于一个时间点 t，定义 t^- 和 t^+ 来区别系统在时间点 t 所发生的事件之前和之后的不同状态：如果一个性质在时间点所发生事件改变其状态之前成立，则称该性质在 t^- 成立；如果一个性质在时间点所发生事件改变其状态之后成立，则称该性质在 t^+ 成立。相应地，一个发生在 t 的事件的含义为该事件发生于 t^- 和 t^+ 之间。例如，如果一个任务实例 J 在 t^- 执行，但是在 t^+ 没

有执行，则可知 J 在 t 时刻放弃执行了（被抢占了或者执行结束了）；如果 J 在 t 时刻开始执行，则可知 J 没有在 t^- 执行但是却在 t^+ 执行。此外，一个性质在 t 成立当且仅当其在 t^- 和 t^+ 都成立。

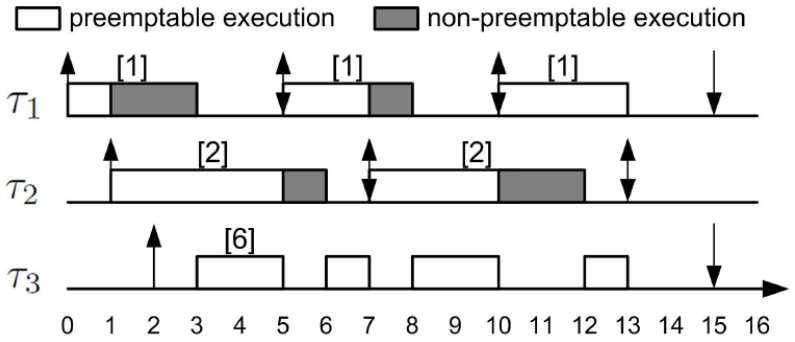
下面使用图 5.3 中的例子来解释 PrtManage(J_i) 是如何工作的。假设 $M=2$，$\Psi_1=\{1,3,5,9\}$，$\Psi_2=\{2,4,7\}$ 以及 $\Psi_3=\{6,8\}$。图 5.3(b)中所示为一个可能的调度序列。任务 τ_1 在 0 时刻释放了第一个任务实例 J_1^1，并将其索引列表 Λ_1 中的第一个索引删除，如图 5.3(c)所示。任务 τ_2 在 1 时刻释放了第一个任务实例 J_2^1。此时又一个空闲处理器，因此 PrtManage 重置所有任务的索引列表，将 J_1^1 变为不可抢占，并删除 Λ_2 中的第一个索引。任务 τ_3 在 2 时刻释放了第一个任务实例 J_3^1，此时所有正在运行的可抢占任务实例中最低的优先级为 2，比 J_3^1 对应的优先级 6 高，因此 J_3^1 获得优先级 6，且 Λ_3 中的第一个索引被删除。在 3 时刻 J_1^1 执行结束，J_3^1 接着获得处理器并开始执行。PrtManage 在 3 时刻并没有被触发执行。在 5 时刻，τ_1 释放了第二个任务实例 J_1^2，此时所有正在运行的可抢占任务实例中最低的优先级为 6，比 J_1^2 对应得优先级 1 低，因此 PrtManage 执行第 7 至第 13 行：另一个正在运行的任务实例 J_2^1 变为不可抢占，RedirectToHigh 将 Λ_2 中的第一个索引由 4 重定向至 2。然后，刚释放的任务 J_1^2 获得优先级 1 且相应的索引被删除。因为此时 J_3^1 为可抢占，J_1^2 将抢占 J_3^1 并开始执行。在 6 时刻 J_2^1 执行结束，J_3^1 活动处理器并恢复执行。7 时刻 τ_2 释放了第二个任务实例 J_2^2，此时所有正在运行的可抢占任务实例中最低的优先级为 6，低于 J_2^2 相对于的优先级 2。因此 PrtManage 执行第 7 至第 13 行：另一个正在运行的任务实例 J_1^2 变为不可抢占，RedirectToHigh 将 Λ_1 中的前两个个索引由 3，5 重定向至 1，3。最后，J_2^1 获得优先级 2 且响应的索引被删除。

运行时一种可能的情况为一个高优先级任务实例 J_h 正好在一个低优先级任务实例 J_l 执行结束时释放。这种情况下，调度器将暂时不让 J_l 释放执行结束的信号，而是被 J_h 抢占，并等到下一个 J_l 被调度执行的时间点在释放执行结束信号。通过这一规定，就排除了一个高优先级任务实正好在一个低优先级任务实例执行结束时开始执行。根据这个性质，可以得到下面的性质（该性质在下一节的可调度性分析中有重要作用）：

Task	C_i	D_i	T_i
τ_1	3	5	5
τ_2	6	7	7
τ_3	6	15	15

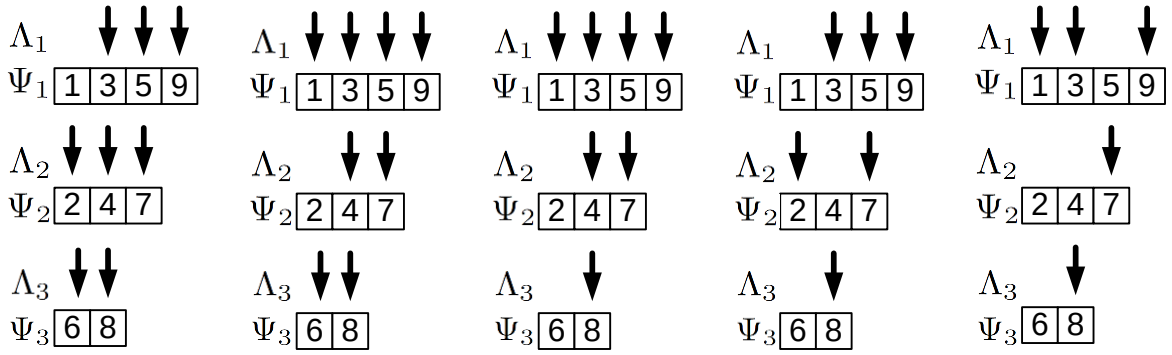
(a) 例子任务集

(a) Task set example



(b) 调度序列（括号中的数字为该任务实例获得的优先级）

(b) Scheduling sequence (the numbers in the bracket denote the priority assigned to the job)



(c) 0^+　(d) 1^+　(e) 2^+　(f) 5^+　(g) 7^+

图 5.3 优先级管理例程 $\mathrm{PrtManage}(J_i)$

Fig. 5.3 The priority management routine $\mathrm{PrtManage}(J_i)$

引理 5.1：令 $t_1,t_2:t_1<t_2$ 为两个满足下述条件的时间点：

- 一个可抢占任务实例 J_1 在 t_1^- 执行，
- 一个满足 $\mathrm{prt}(J_2)\triangleright\mathrm{prt}(J_1)$ 的任务实例 J_2 在 t_2 开始执行。

则在时间区域 $[t_1,t_2]$ 内的某个时间点上一定有一个优先级低于 $\mathrm{prt}(J_2)$ 的任务实例（空闲任务实例）被抢占。

证明：令 t_x 为区域 $[t_1,t_2]$ 内第一个满足如下条件的时间点：一个优先级不低于 $\mathrm{prt}(J_2)$ 的任务实例 J_x 在 t_x 执行。值得注意的是，一定存在一个这样的 t_x。这是因为 J_2 在 t_2 开始执行。r_x 为 J_x 的释放时间。

首先用反正法证明 $r_x=t_x$。假设 $r_x<t_x$，即 J_x 在 t_x^- 之前已经释放了但是在 t_x^- 没有执行。因此在 t_x^- 运行的任何任务实例的优先级都一定高于 $\mathrm{prt}(J_x)$。另一方面，至少有一个优先级低于 J_x 的可抢占任务实例在 t_1^- 执行（J_1），因此在 (t_1,t_x) 内一定存在一个优先级高于 $\mathrm{prt}(J_x)$，因此也就高于 $\mathrm{prt}(J_2)$ 的任务实例开始执行。这与 t_x 的定义相矛盾。因此可证 $r_x=t_x$，即 J_x 在 t_x 时刻释放。根据 t_x 的定义可知在 t_x^- 或之前，所有处理器上执行的任务实例的优先级都都低于 J_2，因此当 J_x 在 t_x 时刻释放时，它一定抢占了某个低优先级任务（可能是空闲任务）并开始执行。需要注意的是，如上文所述，JPA 的运行时调

度会在一个高优先级任务实例在一个低优先级执行结束的同时释放时也触发抢占。综上所述，一定存在某个优先级低于 $\mathrm{prt}(J_2)$ 的任务实例在 t_x 被抢占。　□

5.2　JPA 的可调度性分析

本节将详细介绍 JPA 的设计阶段优先级分配所有的测试条件 $\mathrm{TEST}\left(\tau_k, \{\delta_j\}_{\tau_j \in \tau}\right)$，并证明如果在优先级分配的某一步该测试条件成立，且相应的优先级值 ρ 被加入到了 τ_k 的优先级列表 Ψ_k 中去，则运行时 τ_k 所有被赋予优先级 ρ 的任务实例都能够满足截止期，由此说明如果设计阶段优先级分配成功了，那么任务集是可以被 JPA 调度的。

下面将集中考虑对应于优先级值 ρ 这一步的测试。首先在第 5.2.1 节中来分析任意一个优先级为 ρ 的任务实例 J_k 的运行情况，来找到那些会干涉 J_k 的任务实例。然后在第 5.2.2 节中将上述信息与设计阶段优先级分配相联系，来构造能够保证 J_k 可调度性的测试条件 TEST()。

5.2.1　运行时状态分析

在 JPA 的运行时调度中，一个释放的任务实例获得其相应索引列表中第一个索引所指向的值作为其优先级，然后这第一个索引就被删除。因此，可以将所以列表 Λ_k 看作是对 τ_k 未来任务实例优先级赋值的一个“规划”：如果从某个特定的时间点开始 Λ_k 不再被更新改变的话，则 τ_k 的第 x 个未来任务实例将会获得 Λ_k 中第 x 个索引所指向的优先级值。但是，在某个任务实例释放之前，与之相关的优先级规划可能会因为 PrtManage 中的 RedirectToHigh 操作而不时发生改变。下面定义“期望优先级的”概念，来描述一个任务实例在某个特定的时间点所被“规划”的优先级：$\mathrm{epp}(J,t^-)$（$\mathrm{epp}(J,t^+)$）表示如果从 t^-（t^+）开始所有任务实例都严格按照 t^- 时的索引列表来获取优先级的话，J 在最终释放时将获得的优先级。例如，在图 5.2 所示的例子中 $\mathrm{epp}(J_2^2,0^+)=4$ 以及 $\mathrm{epp}(J_2^2,5^+)$。此外，规定对于 J 释放之后的任何时间点 t 都有 $\mathrm{epp}(J,t^+)=\mathrm{prt}(J)$。

下面考虑一个满足 $\mathrm{prt}(J_k)=\rho$ 的任务实例 J_k。令 t_p 为 J_k 变为不可抢占的时间点。如果 J_k 在整个执行过程都没有变为不可抢占，则令 t_p 等于其完成执行时间 f_k。令 t_s 表示 J_k 开始执行的时间，且可知 $t_s < t_p$。这是因为若 J_k 在 t_p 变为不可抢占其必须在 t_p^- 执行，而 J_k 在 t_s^- 没有执行。下面定义 J_k 的决定时刻 t_0：

定义 5.1（决定时刻 t_0）：J_k 的决定时刻，记为 J_k，使在 t_p 之前满足下列两个性质的最晚时间点：

1. J_k 在 t_0^- 没有执行；

2. 一个优先级低于 $\mathrm{prt}(J_k)$ 的任务实例在 t_0 被抢占。

令 J_0 为在 t_0 被抢占的那个任务实例。如果 t_0 时刻有多个任务实例被抢占，则令 J_0 为其中优先级最低的那个。需要注意，一定存在一个这样的 t_0，因为在初始时刻 0 定义 5.1 中的两个条件都满足。称时间区域 (t_0,t_p) 为问题窗口。决定时刻 t_0 有以下重要性质：

引理 5.2：$\mathrm{epp}\left(J_k,t_0^+\right)=\mathrm{prt}\left(J_k\right)$

证明：用反正法证明，假设 $\mathrm{epp}\left(J_k,t_0^+\right)\lhd\mathrm{prt}\left(J_k\right)$，则在时间区域 (t_0,r_k) 内一定存在一个时间点 t 使得 J_i 在 t 时刻期望优先级变高。根据 JPA 的优先级管理规则可知在 t 时刻一定有某个优先级低于 $\mathrm{epp}(J_k,t^-)$（因此也低于 $\mathrm{prt}(J_k)$）的任务实例被抢占。此外还知道 J_k 在 t^- 没有执行（因为只有在 J_k 释放前 $\mathrm{epp}(J_k,t_0^+)$ 才会变高）。因此 t 是一个 t_0 之后同时满足定义 5.1 中两个条件的时间点，这与 t_0 的定义相矛盾，由此引理可证。　□

根据引理 5.2 可知 t_0 是那个最终决定了 J_k 优先级的时间点（这就是为什么其被称为决定时刻）。下面的分析主要是找出那些在问题窗口 (t_0,t_p) 里能够干涉 J_k 的任务集，然后在第 5.3.2 节中，这些信息将被用来推出 JPA 设计阶段优先级分配中对应于 ρ，也就是 $\mathrm{prt}\left(J_k\right)$ 的可调度性判定条件。

因为那些与 J_k 并行执行的工作量不能对 J_k 造成干涉，所以只有当一个任务实例在 (t_0,t_p) 中某个 J_k 没有在执行的时间点上执行时，才会对 J_k 产生干涉。下面第一步是证明这样的任务实例如果在 t_0^+ 是可抢占的，则其优先级一定要高于 $\mathrm{prt}(J_k)$，如下引理所述：

引理 5.3：令 t 为 (t_0,t_p) 内一个时间点，且 J_k 在 t^+ 没有执行。令 J_i 为另一个任务 $\tau_i\left(\neq\tau_k\right)$ 的一个实例，且 J_i 在 t^- 执行。如果 J_i 在 t_0^+ 还未释放或者已经释放了且为可抢占的，则一定满足：

$$\mathrm{prt}\left(J_i\right)\rhd\mathrm{prt}\left(J_k\right)$$

证明：用反正法证明，假设

$$\mathrm{prt}\left(J_i\right)\lhd\mathrm{prt}\left(J_k\right) \tag{5.1}$$

注意因为 $\tau_i\neq\tau_k$，所以可知 $\mathrm{prt}\left(J_i\right)\neq\mathrm{prt}\left(J_k\right)$（不同任务的优先级列表里没有重叠的优先级值），因此 $\mathrm{prt}\left(J_i\right)\rhd\mathrm{prt}\left(J_k\right)$ 的否定直接为 $\mathrm{prt}\left(J_i\right)\lhd\mathrm{prt}\left(J_k\right)$。

因为 J_i 在 t_0^+ 还未释放或者已经释放了且未可抢占的，它在 (t_0,t_p) 要么变为不可抢占的，要么一直保持可抢占。定义时间点 t'：

- $t'=t$，如果 J_i 一直保持可抢占；
- 否则 t' 为 (t_0,t_p) 内 J_i 变为可抢占的时间点。

根据这个定义，可知 J_i 在 t'^- 一定是可抢占的。下面讨论两种不同的情况：

1. $t' < r_k$。在这种情况下，根据$t' < r_k$和$r_k \le t_s$可知$t' < t_s$（t_s是J_k的开始执行时间）。关于时间区域$[t', t_s]$可知 i)一个可抢占任务实例J_i在t'^-执行，ii)一个优先级高于$\mathrm{prt}(J_i)$的任务实例J_k在t_s开始执行。因此根据引理 5.2 可知$[t', t_s]$中一定存在一个时间点t''，使得某个优先级低于$\mathrm{prt}(J_k)$的任务实例在t''被抢占。因为$t'' \in [t', t_s]$，又可知J_k在t_s^-没有执行。综上所述，t''满足定义 5.1 中的两个条件，又知$t'' \in (t_0, t_p)$，这与t_0的定义相矛盾。

2. $t' \ge r_k$。因为在t'^+ J_k已经就绪但是还没有开始执行，可知在t'^+执行的所有任务实例要么比J_k的优先级高，要么是不可抢占的。又因为J_i在t'^-是可抢占的（见上文讨论），可知J_i在t'^-执行且在t变为不可抢占。因为$\mathrm{prt}(J_i) \lhd \mathrm{prt}(J_k)$，可知$J_k$也在$t'$执行且$t'$时刻发生的抢占会使$J_k$变得不可调度（或者$J_k$在$t'$之前已经变为不可调度了），这与$t_p > t_s \ge t'$相矛盾。

综上所述，两种情况都会导致矛盾，由此引理可证。　□

上述引理提供了关于哪些任务实例在(t_0, t_p)内干涉J_k的初步信息。下一步将把这个信息以任务实例在t_0的期望优先级的形式表达出来（仅针对那些在t_0后释放的任务实例）：

引理 5.4：令t为(t_0, t_p)内一个时间点且J_k在t^+没有执行。令J_i为另一个任务$\tau_i (\ne \tau_k)$的一个实例，且J_i在t^-执行。如果J_i在t_0^+还未释放，则一定满足：

$$\mathrm{epp}\left(J_i, t_0^+\right) \rhd \mathrm{prt}\left(J_k\right) \tag{5.2}$$

证明：用反正法证明，假设

$$\mathrm{epp}\left(J_i, t_0^+\right) \lhd \mathrm{prt}\left(J_k\right) \tag{5.3}$$

注意因为$J_i \ne J_k$所有一定有$\mathrm{epp}(J_i, t_0^+) \ne \mathrm{prt}(J_k)$。首先，因为$J_i$不是在$t_0$之前释放的，根据引理 5.3 可知$\mathrm{prt}(J_i) \rhd \mathrm{prt}(J_k)$。将其应用于(5.3)，可得

$$\mathrm{prt}\left(J_i\right) \rhd \mathrm{epp}\left(J_i, t_0^+\right)$$

即J_i的期望优先级在t_0后的某个时间点变高了。

令t'为t_0之后第一个满足下述条件的时间点：J_i的期望优先级在t'变高了。因此可知一定存在某个优先级低于$\mathrm{epp}(J_i, t_0^+)$的任务实例在t'被抢占了，又根据(5.3)可知这个被抢占任务实例的优先级低于$\mathrm{prt}\left(J_k\right)$。下面讨论两种情况：

1. J_k在t'^-执行。因为一个优先级低于$\mathrm{prt}(J_k)$的任务实例在t'被抢占了，根据 JPA 的优先级管理规则可知J_k在t'变为不可抢占，即$t' = t_p$。另一方面，因为J_i在t执行切一个任务实例在释放以后其期望优先级不会再变高了，因此可知$t' \le t$，又

因为$t \in (t_0, t_p)$可知$t' < t_p$，这与上面所示的$t' = t_p$相矛盾。

2. J_k在t'^-没有执行。这种情况下，t'是t_0之后同时满足以下两个条件的时间点：

1) J_k在t'^-没有执行；

2) 一个优先级低于$\mathrm{prt}(J_k)$的任务实例在t'^-被抢占。

也就是说，t'同时满足定义 5.1 中的两条件，这与t_0的定义相矛盾。

综上所述，两种情况都将导致矛盾，由此引理可证。　□

上述引理找到了那些所以在t_0或t_0后释放的任务实例中可能在问题窗口里干涉J_k的任务实例。下面的引理将处理那些在t_0之前释放的任务实例：

引理 5.5：令t为(t_0, t_p)内一个时间点且J_k在t^+没有执行。令J_i为另一个任务$\tau_i (\neq \tau_k)$的一个实例，且J_i在t^-执行。如果J_i在t_0之前已经被释放了，则J_i在t_0^+一定是不可抢占的。

证明：用反证法证明，假设J_i在t_0^+为可抢占的。根据引理 5.3 可知：

$$\mathrm{prt}(J_i) \rhd \mathrm{prt}(J_k) \tag{5.4}$$

因为J_0是t_0时刻所有被抢占任务实例中优先级最低的那个，且根据t_0的定义可知$\mathrm{prt}(J_k) \rhd \mathrm{prt}(J_0)$。将此与(5.4)结合可知$\mathrm{prt}(J_i) \rhd \mathrm{prt}(J_0)$。因此$J_i$在$t_0^-$也在执行，根据 JPA 的优先级管理规则可知在$J_0$时刻$J_0$被抢占时$J_i$将变为不可抢占，这与假设相矛盾，由此引理可证。　□

5.2.2 设计阶段可调度性分析

上一节中找到了那些在问题窗口(t_0, t_p)中可能干涉J_k的任务实例。在这一节中，将把这个信息和设计阶段的优先级分配过程中的信息联系起来，以构建保证J_k可调度性的测试条件$\mathrm{TEST}\left(\tau_k, \{\delta_j\}_{\tau_j \in \tau}\right)$。这是为在对优先级值$\rho$得时候进行的测试，其中每个任务$\tau_j$的$\delta_j$表示到目前为止剩下的需要加入$\tau_j$的优先级列表$\Psi_j$中的优先级值个数，也就是$\Psi_j$中优先级高于$\rho$的优先级值的个数。

可调度性判定的总体框架如下：假设任务集τ在运行时不可调度，J_k是运行时第一个错失截止期的任务实例J_k。然后求出任务集在问题窗口(t_0, t_p)内工作量的总和，这个总和应该大于问题窗口中所有处理器在J_k没有运行时所提供的处理器能力。取上述命题的逆否命题，可以得到一个保证J_k可调度性的充分判定条件，将此条件应用于任务集中的每个任务，便可得到一个保证整个任务集可调度性的充分判定条件。

（1）工作量

令$\ell = t_p - t_0$。第一步是求出每个任务在(t_0, t_p)内工作量的上限。与前两章类似，τ_i在

(t_0,t_p)内的工作量可以分为三部分：（1）前部任务实例的工作量；（2）中部任务实例的工作量；（3）后部任务实例的工作量。且定义两类τ_i在长度为ℓ的时间区域内的工作量上限：

- $W^{\mathrm{nc}}(\tau_i,\ell)$：$\tau_i$的工作量上限，如果其没有前部任务；
- $W^{\mathrm{ci}}(\tau_i,\ell)$：$\tau_i$的工作量上限，如果其有前部任务。

可以通过与前两章相似的方法计算这两个上限：

$$W^{\mathrm{nc}}(\tau_i,\ell)=\left\lfloor\frac{\ell}{T_i}\right\rfloor C_i+\min\left(C_i,\ell \bmod T_i\right)$$

$$W^{\mathrm{ci}}(\tau_i,\ell)=\begin{cases}\ell & \ell<C_i\\ \left\lfloor\dfrac{\ell-C_i}{T_i}\right\rfloor C_i+C_i+\alpha(\ell) & \ell\geq C_i\end{cases}$$

其中$\alpha(\ell)$定义为：

$$\alpha(\ell)=\min\left(C_i,\max\left(0,(\ell-C_i)\bmod T_i-(T_i-D_i)\right)\right) \tag{5.5}$$

（2）干涉

令c'为J_k在(t_0,t_p)内的执行时间（即J_k变为不可抢占之前的执行时间），进而C_k-c'为J_k变为不可抢占之后的执行时间。如第 5.3.1 节中所述，τ_i的工作量中与J_k并行执行的部分不能够阻止J_k执行。与工作量上限相似，定义两类干涉上限：

- $I_k^{\mathrm{nc}}(\tau_i,\ell,c')$：$\tau_i$的干涉上限，如果其没有前部任务；
- $I_k^{\mathrm{ci}}(\tau_i,\ell,c')$：$\tau_i$的干涉上限，如果其有前部任务。

根据引理 5.4 可知，任务τ_i在(t_0,t_p)内任意一个J_k没有在执行的时间点执行的任意一个中部或后部任务实例J_i都满足

$$\mathrm{epp}\left(J_i,t_0^+\right)\rhd\rho \tag{5.6}$$

其中$\rho=\mathrm{prt}(J_k)$。就是说，τ_i所有能够干涉J_k的中部或后部任务实例的优先级都高于ρ。这个命题对τ_k自己的任务实例同样成立，这是因为根据引理 5.2 可知$\mathrm{epp}(J_k,t_0^+)\ngtr\rho$，又因为$\tau_k$得任何一个在$t_0$之后$J_k$的释放之前释放的任务实例$J_k'$都满足$\mathrm{epp}(J_k',t_0^+)\rhd\mathrm{epp}(J_k,t_0^+)$，因此可知$\mathrm{epp}(J_k',t_0^+)\rhd\rho$。另一方面，根据 JPA 的设计阶段优先级分配规则，Ψ_i中比ρ优先级高的数值为在分配优先级ρ这一步时的δ_j（对于τ_k为δ_k-1因为δ_k中包括了ρ）。因此可知：

$$I_k^{\mathrm{nc}}(\tau_i,\ell,c')\leq\begin{cases}\delta_i\times C_i & i\neq k\\ (\delta_i-1)\times C_i & i=k\end{cases}$$

因为任务集为限制截止期，一个任务在同一时刻最多有一个活跃任务实例，因此每个任务最多有一个前部任务。如果τ_i有前部任务，则其干涉的总和上限不超过：

$$I_k^{\mathrm{ci}}(\tau_i,\ell,c') \le \begin{cases} (\delta_i+1)\times C_i & i \ne k \\ \delta_i \times C_i & i = k \end{cases}$$

此外，因为一个任务的干涉之能发生在J_k不执行的时候，因此$I_k^{\mathrm{nc}}(\tau_i,\ell,c')$和$I_k^{\mathrm{ci}}(\tau_i,\ell,c')$都不能超过$\ell - c'$。计算$\tau_k$自己在问题窗口内的干涉，只需要考虑其在$J_k$之前释放的任务实例。令$t_d$为$J_k$之前最后一个释放的实例的绝对截止期，则可知：

$$t_d \le r_k - (T_i - D_i)$$

因为假设J_k错失其截止期，则一定有

$$t_p > (r_k + D_k) - (C_k - c')$$

将上述两式结合可得

$$t_p > t_d + T_i - C_k + c'$$

因此t_0与t_d之间的时间距离至多为$\ell - (T_i - C_k + c')$。

如果τ_k没有前部任务实例，则其在长度为x的时间区域内的干涉可以通过需求上限函数$\mathrm{DBF}(\tau_k,x)$来计算[129]。$\mathrm{DBF}(\tau_k,x)$为任务τ_k在长度为x的时间区域内所有释放时间和绝对截止期都在此区域内的任务实例的工作量总和：

$$\mathrm{DBF}(\tau_k,x) = \left\lfloor \frac{x - D_i}{T_i} + 1 \right\rfloor \times C_i$$

如果τ_k有前部任务实例，它的干涉通过将前部任务工作量加入需求上限函数来计算，如[34]中所示：

$$\mathrm{DBF}^{\mathrm{ci}}(\tau_k,x) = \left\lfloor \frac{x}{T_i} \right\rfloor \times C_i + \min(C_i, x \bmod T_i)$$

综上所述，可以得到τ_i在问题窗口内的干涉上限如下：

$$I_k^{\mathrm{nc}}(\tau_i,\ell,c') = \begin{cases} \min\{W^{\mathrm{nc}}(\tau_i,\ell), \delta_i \cdot C_i, \ell - c'\} & i \ne k \\ \min\{\mathrm{DBF}(\tau_i, \ell - (T_i - C_k + c')), (\delta_i - 1)C_i\} & i = k \end{cases}$$

$$I_k^{\mathrm{ci}}(\tau_i,\ell,c') = \begin{cases} \min\{W^{\mathrm{ci}}(\tau_i,\ell), (\delta_i+1)C_i, \ell - c'\} & i \ne k \\ \min\{\mathrm{DBF}^{\mathrm{ci}}(\tau_i, \ell - (T_i - C_k + c')), \delta_i \cdot C_i\} & i = k \end{cases}$$

根据引理 5.5 可知所有的前部子任务在t_0^+都是不可抢占的。另一方面，还可知在t_0^+最多有$M-1$个不可抢占的任务实例。这是因为，首先任意时刻至多有M个不可抢占任务实例，其次在t_0^+一定有一个任务实例刚刚开始执行（即那个在t_0抢占了J_0的任务实例）。

根据上面的讨论可知，用通过第 3 章中图 3.4 中的算法可以求得到整个任务集在 (t_0,t_p) 内干涉总和上限 $\Omega_k\left(\ell,c'\right)$。

此外，对 Ω_k 可知如下性质：

引理 5.6：给定一个 $z:0\le z\le c$，令 $x^*=x-z$ 且 $c^*=c-z$，则一定有

$$\Omega_k\left(x,c\right)\ge\Omega_k\left(x^*,c^*\right)$$

证明：根据 I_k^{nc} 和 I_k^{ci} 的定义可知

$$I_k^{\mathrm{nc}}\left(\tau_i,x,c\right)\ge I_k^{\mathrm{nc}}\left(\tau_i,x^*,c^*\right)$$

$$I_k^{\mathrm{ci}}\left(\tau_i,x,c\right)\ge I_k^{\mathrm{ci}}\left(\tau_i,x^*,c^*\right)$$

注意其中 $x-c=x^*-c^*$。令 τ^{nc} 为最大化 $\Omega_k(x,c)$ 中所有没有前部实例任务的集合和所以有前部实例任务的集合，则有：

$$\begin{aligned}\Omega_k\left(x,c\right)&\ge\sum_{\tau_i\in\tau^{\mathrm{nc}}}I_k^{\mathrm{nc}}\left(\tau_i,x,c\right)+\sum_{\tau_i\in\tau^{\mathrm{ci}}}I_k^{\mathrm{ci}}\left(\tau_i,x,c\right)\\&\ge\sum_{\tau_i\in\tau^{\mathrm{nc}}}I_k^{\mathrm{nc}}\left(\tau_i,x^*,c^*\right)+\sum_{\tau_i\in\tau^{\mathrm{ci}}}I_k^{\mathrm{ci}}\left(\tau_i,x^*,c^*\right)\\&=\Omega_k\left(x^*,c^*\right)\end{aligned}$$

由此引理可证。 □

（3）可调度性判定条件

首先介绍在给定 t_0 和 c' 的情况下的可调度性判定条件。

引理 5.7：令 c' 为 J_k 在变为不可抢占之前的执行时间，且 $c''=C_k-c'$ 为 J_k 变为不可抢占之后的执行时间。如果满足条件

$$\Omega_k\left(d_k-t_0-c'',c'\right)<\left(d_k-t_0-C_k\right)\times M \tag{5.7}$$

则 J_k 一定可以满足截止期。

证明：用反证法证明，假设 J_k 满足条件(5.7)但是不能满足其截止期。因为一个任务实例在变为不可抢占后便不能受到其它任务干扰而可以一直执行至结束，因此 J_k 一定在 d_k-c'' 之后变为不可抢占的，即 $t_p>d_k-c''$。另一方面，根据(5.7)可知整个任务集的干涉总和不足以使 $\left(t_0,d_k-c''\right)$ 内所有 J_k 没有执行的时间点上让所有处理器都一直忙碌（其中 $d_k-t_0-C_k=d_k-c''-t_0-c''$）。因此 J_k 在 d_k-c'' 之前的执行时间一定严格大于 c'，这与 $t_p>d_k-c''$ 相矛盾。由此引理可证。 □

现在将介绍本节的主要结果，即 JPA 可调度性判定的一般形式：

定理 5.1：JPA 的设计阶段优先级分配中的测试条件定义为：

$$\mathrm{TEST}\left(\tau_k,\left(\delta_j\right)_{\tau_j\in\tau}\right)\Box\left(\forall x\in\left[X_l,X_u\right]:\Omega_k\left(x,C_k\right)<\left(x-C_k\right)\times M\right) \tag{5.8}$$

其中

$$X_l=\left(\delta_k-1\right)T_k+D_k$$

$$X_u=\frac{\left(\sum_{\tau_i\in\tau}\left(\delta_i+1\right)\times C_i-C_k\right)}{M}+C_k$$

则任何可以成功完成JPA的设计阶段优先级分配的任务集都可以被JPA的运行时调度算法成功调度。

证明：根据引理 5.6 可知

$$\Omega_k\left(x,c\right)<\left(x-C_k\right)\times M\Rightarrow\Omega_k\left(x-z,c-z\right)<\left(x-C_k\right)\times M$$

将此结论和$c'+c''=C_k$一起应用于(5.8)可得：

$$\Omega_k\left(x-c'',c'\right)<\left(x-C_k\right)M \tag{5.9}$$

因此根据引理 5.7 可知对于$x=d_k-t_0$，J_k一定是可调度的。下面将证明如果J_k错失截止期，则d_k-t_0一定是在$\left[X_l,X_u\right]$范围内。根据引理 5.2 可知J_k的优先级ρ等于$\mathrm{epp}(J_k,t_0^+)$。因此，从t_0开始，τ_k将要释放那些具有Ψ_k中更高优先级的任务实例（直到J_k被释放）。根据 JPA 的优先级分配规则可知，Ψ_k中比ρ高的优先级有δ_k-1。因此，J_k至少是在$t_0+\left(\delta_k-1\right)T_k$，即$d_k-t_0\geq\left(\delta_k-1\right)T_k+D_k$。下面将证明$d_k-t_0<X_u$。上文中已经证明一个任务$\tau_i$在$(t_0,t_p)$中$J_k$没有执行的时间点上所能执行的任务个数最多为$\delta_i+1$（其中$\delta_i$个在$t_0$之后释放，一个在$t_0$之前释放）。因此，$(t_0,t_p)$中$J_k$没有执行的时间点的工作量总和不超过

$$\frac{\sum_{\tau_i\in\tau}\left(\delta_i+1\right)\times C_i-C_k}{M}$$

因此J_k的执行结束时间不超过

$$t_0+\frac{\left(\sum_{\tau_i\in\tau}\left(\delta_i+1\right)\times C_i-C_k\right)}{M}+C_k$$

因为J_k他的执行结束时间一定大于d_k，所以最终可得$d_k-t_0<X_u$。　□

在任务集所有参数都为整数的情况下，只需要对$\left[X_l,X_u\right]$中的每个整数值进行测试，因此可调度性判定条件(5.8)为伪多项式复杂度。

5.3　运行时效率优化

对于每个任务τ_i，在运行时 PrtManage 都维护一个优先级列表Ψ_i和一个索引列表Λ_i。这两个数据结构的空间复杂度都为伪多项式的。对于索引列表Λ_i操作的时间复杂

度也是伪多项式的。这对于某系对运行时开销和存储资源敏感的嵌入式系统可能是不可接受的。下面将介绍一个 JPA 的改进版本 JPAZ。JPAZ 在运行时只使用常数大小的优先级列表和索引列表，因此效率得到大幅度提高。与此同时，JPAZ 具有和 JPA 几乎一样好的实时性能。

5.3.1 $\mathrm{JPA_Z}$

调度算法 $\mathrm{JPA_Z}$ 对应于一个可有系统设计者选择的常数 Z。在设计阶段的优先级分配过程中，$\mathrm{JPA_Z}$ 只为一个长度为 $Z\times T_{\min}$ 的时间区域内的所能释放的任务实例构建优先级列表，其中 $T_{\min}$ 为所有任务中的最小周期（JPA 中为一个长度为 UB 最大的忙碌期内的所能释放的任务实例构建优先级列表）。因此，每个任务的优先级列表（及索引列表）中元素个数的上限为常数 Z。

在运行时，$\mathrm{JPA_Z}$ 使用与 JPA 相同的方式来管理和分配优先级。唯一的区别是索引列表 Λ_i 中的索引可能不足以覆盖任务 τ_i 在一个忙碌期内所有释放的任务实例。因此可能在某个任务实例释放时对应的索引列表已经空了，因此无法根据索引列表来获取优先级。在种情况下，被释放的任务实例将使用获得最低优先级 $\perp$。换言之，所有不能从索引列表来获取优先级的任务实例（称为底部任务实例）都具有相同的最低优先级，而这比所有根据索引目录获得优先级的任务实例（称为普通任务实例）的优先级都低。

$\mathrm{JPA_Z}$ 背后的主要思想是，在一般情况下，只有那些离忙碌期起始点较近的任务实例更容易措施截止期。因为任务的资源利用率总和小于 M，所以随着忙碌期不断延长，处理器所能提供的处理能力与任务集的工作量总和之间的差距越来越大，因此那些远离忙碌期起始点的任务实例比较通常不会措施截止期。因此在 $\mathrm{JPA_Z}$ 中，对任务实例优先级顺序的挖掘仅限于那些离忙碌期起始点较近的任务实例，而其它的任务实例即使用比较简单的方式进行调度也不容易措施截止期。

5.3.2 可调度性分析

与 JPA 相似，$\mathrm{JPA_Z}$ 设计阶段优先级分配中的测试条件 TEST() 能够保证所有普通任务实例的可调度性。下面只需考虑，在所有普通任务实例都满足截止期的前提下，底部任务实例是否也都能满足截止期。

假设任务底部可调度，令任务 τ_k 的实例 J_k 为第一错失截止期的底部任务实例。令 t_0 为 r_k 之前存在处理器空闲的最晚时间点。令 $\chi = d_k - t_0$。一个任务 τ_i 在时间区域 $[t_0, d_0)$ 内的工作量总和的上限为 $W^{\mathrm{nc}}(\tau_i, \chi)$（如果 τ_i 没有前部任务实例）或者 $W^{\mathrm{ci}}(\tau_i, \chi)$（如果 τ_i 有前部任务实例）。如果 $i = k$，只有在 J_k 之前释放的任务可能造成干涉，因此可以使用

$\mathrm{DBF}(\tau_i, \chi - T_k)$（如果$\tau_i$没有前部任务实例）或者$\mathrm{DBF}^{\mathrm{ci}}(\tau_i, \chi - T_k)$（如果$\tau_i$有前部任务实例）来作为其干涉的上限。因此每个任务的干涉上限为：

$$I_k^{\mathrm{nc}}(\tau_i, \chi) = \begin{cases} \min\{W^{\mathrm{nc}}(\tau_i, \chi), \chi - C_k\} & i \neq k \\ \mathrm{DBF}(\tau_i, \chi - T_k) & i = k \end{cases}$$

$$I_k^{\mathrm{ci}}(\tau_i, \chi) = \begin{cases} \min\{W^{\mathrm{ci}}(\tau_i, \chi), \chi - C_k\} & i \neq k \\ \mathrm{DBF}^{\mathrm{ci}}(\tau_i, \chi - T_k) & i = k \end{cases}$$

根据t_0的定义，可知至多$M-1$个任务有前部任务实例，因此可以使用第 3 章中的图 3.4 中的算法可以求得到整个任务集在$[t_0, d_0)$内干涉总和上限$\Phi_k(\chi)$。因此，对于一个特定的χ，如果条件$\Phi_k(\chi) < (\chi - C_k) \times M$成立，则任务实例$J_k$一定是可调度的。因为$\chi$是未知的，所以需要对所有可能的$\chi$考查上述条件。因为$UB$是忙碌期的最大长度上限，因此可知$\chi \leq \mathrm{UB}$。令一方面，因为$J_k$是底部任务实例，可知$r_k - t_0 \geq Z \times T_{\min}$，即$\chi \geq Z \times T_{\min} + D_k$。根据上述讨论可以得到$\mathrm{JPA}_Z$的可调度性判定如下：

定理 5.2：如果一个任务集能够成功完成JPA_Z设计阶段的优先级任务分配，且满足条件

$$\forall \chi \in [Z \times T_{\min} + D_k, \mathrm{UB}] : \Phi_k(\chi) < (\chi - C_k) \times M \tag{5.10}$$

则该任务集一定可以被JPA_Z调度。

在上述定理中，需要考虑的χ的最小值为$Z \times T_{\min} + D_k$，这是因为如果一个忙碌期太短，期间释放的任务实例都能够从优先级列表获得优先级值而不会是底部优先级实例。

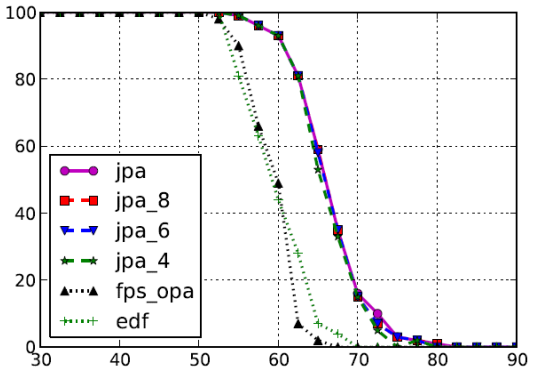
在任务参数都是整数的前提下，只需要对$[Z \times T_{\min} + D_k, \mathrm{UB}]$内的整数进行(5.10)的判定，因此其计算时间复杂度为伪多项式的。另外，JPA_Z设计阶段的优先级任务分配的计算时间复杂度也是伪多项式的，以此JPA_Z可调度性判定总的时间复杂度也是伪多项式的。

5.4　质量评价

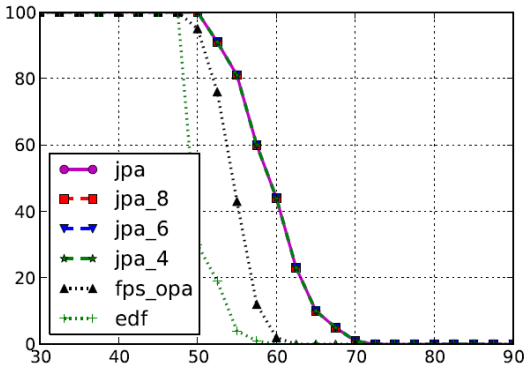
本节对所提出的算法和原有工作中最好的全局固定优先级和 EDF 调度的可调度性分析的接受率进行比较：

- JPA，第 5.1 节中介绍的调度算法，在图中表示为“jpa”。
- JPA_Z，第 5.3 节中介绍的改进运行时效率的调度算法。将常数 Z 分别设为 4，6 和 8，对应的算法在图中分别表示为“jpa-4”，“jpa-6”和“jpa-8”。
- EDF，使用文献[34]的可调度性判定条件，在图中表示为“edf”。

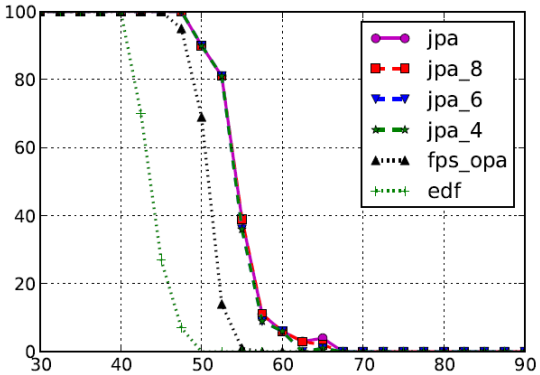
- FPS，使用文献[96]中最优优先级分配（OPA）的全局固定任务优先级调度算法，在图中表示为“fps-opa”。



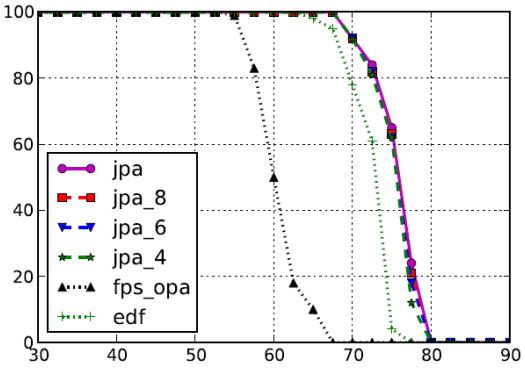
(a) $M=4$，$U_i \in [0.1,0.6]$



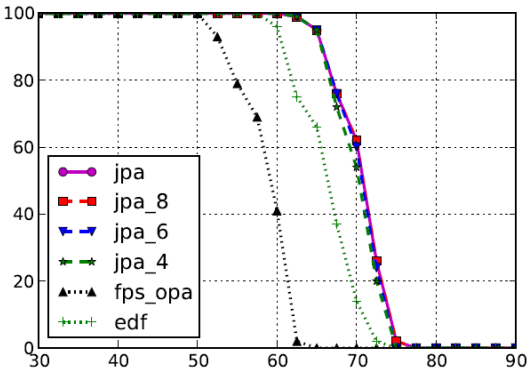
(b) $M=4$，$U_i \in [0.1,0.6]$



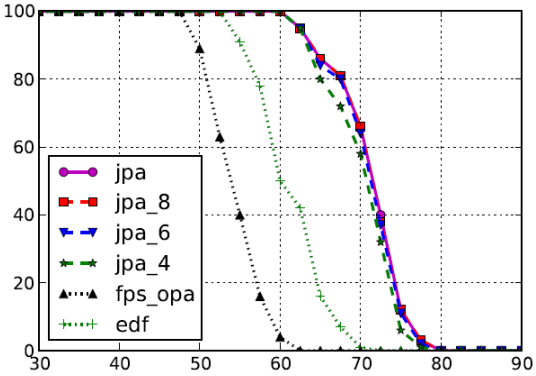
(c) $M=4$，$U_i \in [0.1,0.6]$



(d) $M=4$，$U_i \in [0.1,0.6]$



(e) $M=4$，$U_i \in [0.1,0.6]$



(f) $M=4$，$U_i \in [0.1,0.6]$

图 5.4 实验结果

Fig. 5.4 Experiment results

图 5.4 中所示为在不同正规化资源利用率下各个算法的接受率比较。图 5.4(a)中的参数设置如下：处理器个数 $M=4$，每个任务 τ_i 的 $T_i = D_i$，T_i 在范围 $[10,20]$ 内随机选取，U_i 在范围 $[0.1,0.6]$ 内随机选取。在图 5.4(b)至图 5.4(f)中变化 M 和 U_i 的范围。图 X 轴为任务集的正规化资源利用率总和，Y 轴为接受率（都以百分数的形式）。

实验结果表明，本章提出的算法质量明显优于 FPS 和 EDF。尽管 FPS 和 EDF 的实

时性能在不同的参数下互有优劣，JPA 和JPA_Z始终表现出比 FPS 和 EDF 好的性能。其中，JPA_Z的性能和 JPA 非常接近。即使在 Z 为很小的常数时（例如$Z=4$，这时每个任务的优先级列表和索引列表中只有 4 个元素，因此JPA_Z的运行效率非常高），JPA_Z和 JPA 的性能也几乎没有区别。由此可见，全局过处理器调度中任务的可调度性的确只取决于在一个忙碌期刚刚开始阶段的几个任务实例，而在那之后，即使用非常简单的调度方法也几乎不会出现错失截止期的情况。

5.5　小结

本章提出了一个全局固定实例优先级调度算法 JPA。JPA 通过在任务实例级别挖掘它们的优先级顺序，能够显著地提高全局调度的质量。本章还提出了 JPA 的一个变种JPA_Z。与 JPA 相比，JPA_Z具有非常高的运行时效率，但是却几乎具有和 JPA 一样好的实时性能。通过使用随机生成任务集的实验表明，本章提出的算法与现有的标准全局固定优先级和 EDF 调度算法相比，其实时性能得到大幅度的提高。其中具有较高运行效率的算法JPA_Z，即使在 Z 为很小的常数时也可以获得和 JPA 几乎一样好的接受率。

第6章 准划分固定优先级算法的Liu&Layland资源利用率界限

资源利用率界限的概念最早是在 Liu 和 Laylang 在 1973 年发表的经典文章[14]提出的。如第 2 章所述，资源里用率界限即可以作为一种非常高效的判断任务集可调度性的实用方法，也可以作为衡量调度算法质量的重要指标。[34]中证明了在单处理机调度中，最优的固定优先级调度算法 RMS（单调速率调度）的资源利用率界限为 $N(2^{1/N}-1)$，称为 Liu&Layland 资源利用率界限。为了简化表达，本章中令 $\Theta(N)=N(2^{1/N}-1)$。

多处理机调度一般可以分为两大类：全局调度和划分调度。目前为止所知最好的全局固定优先级调度算法的资源利用率界限为 38%。而划分调度由于受到与背包问题类似的限制，其最坏情况下的资源利用率界限不可能超过 50%。为了突破这一个限制，研究者们提出了准划分调度算法。原有工作中，所有基于优先级的多处理机调度种具有最高资源利用率界限的算法为[54]中所提出的一个准划分固定优先级调度算法，其资源利用率界限为 65%，这仍旧低于 Liu&Layland 资源利用率界限。尤其当任务集中任务个数较少时，两者的差距更大。

本章提出一个新的准划分固定优先级调度算法，这个算法具有 Liu&Layland 资源利用率界限 $\Theta(N)$。这一结果使得著名的 Liu&Layland 资源利用率界限终于被推广到多处理机调度。本章提出的算法使用 RMS 作为每个处理器上的局部调度算法，而且与原有的准划分调度算法具有相同的任务切分开销。

本章首先提出一个算法 RMTS-1，该算法对所有仅含有资源利用率不超过 $\Theta(N)/(\Theta(N)+1)$ 任务的任务集都具有 Liu&Layland 资源利用率界限 $\Theta(N)$。该算法的过程非常简单：将所有任务按照优先级由低到高（即周期由大到小）的顺序进行处理器分配，且在分配的每一步总是选择到目前为止所被分配的负载最低的处理器。然后提出第二个算法 RMTS-2，通过引入一个任务预分配机制来去掉对每个任务资源利用率大小的限制，使得算法对任意的任务集都具有资源利用率界限 $\Theta(N)$。

6.1 基本概念

由于资源利用率界限是针对隐式截止期任务集的特有概念，本章假设只考虑隐式截止期任务集。但是本章提出的算法和其资源利用率界限同样可以通过密度界限的方式适用于限制截止期任务集和任意截止期任务集。假设多处理机平台由 M 个相同的处理器

$\{P_1, P_2, ..., P_M\}$ 组成。下面首先回顾一下 Liu 和 Laylang 的经典结果[14]：

定理 6.1：在单处理机系统上，如果一个任务集 τ 满足条件

$$U(\tau) \le N\left(2^{\frac{1}{N}} - 1\right)$$

则它是可被单调速率算法（RMS）调度的。

$N(2^{1/N} - 1)$ 也被称为 Liu&Layland 资源利用率界限。本章所提出的准划分调度算法的资源利用率结果是建立在上述结果之上的。下面正式定义一下 Liu&Layland 资源利用率界限的简写符号：

$$\Theta(N) = N\left(2^{\frac{1}{N}} - 1\right) \tag{6.1}$$

为了简化叙述过程，本章假设任务集 τ 中的每个任务 τ_i 的资源利用率 U_i 都不超过 $\Theta(N)$。该假设并不影响本章的资源利用率界限结果应用于含有资源利用率高于 $\Theta(N)$ 的任务的任务集：如果一个任务集的正规化资源利用率总和 $U(\tau) \le \Theta(N)$ 且含有资源利用率大于 $\Theta(N)$ 的任务，那么可以将这些任务每一个单独分配到一个处理器上，而让剩下的任务在剩下的处理器上使用本章提出的算法进行分配和调度；如果可以证明这些剩下的任务在剩下的处理器上是可调度的，那么对于整个任务集，资源利用率界限 $\Theta(N)$ 依然成立。

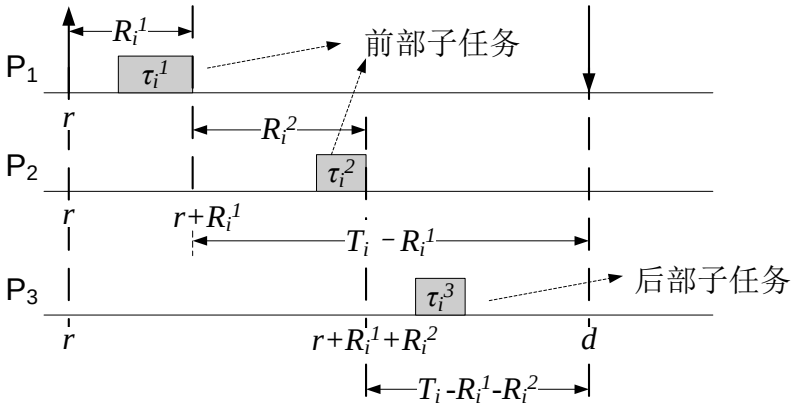


图 6.1 子任务示意图

Fig. 6.1 Illustration of subtasks

一个准划分多处理机调度算法包含两个部分：（1）划分算法；（2）调度算法。其中，划分算法决定如何切分任务，以及如何将每个任务（或者一个任务的每个子任务）分配到一个固定的处理器上。而调度算法决定在运行时在每个处理器上何时执行某个任务（或一个任务的一个子任务）。

在划分算法中，大部分的任务将会被整个分配到某个处理器上，且运行时之在这个

特定的处理器上运行。本章称这类任务为非切分任务。其它的任务被称为切分任务。每个切分任务将被切分成若干个子任务。一个任务的各个子任务将被分配到不同的处理器上，且每个子任务运行时只在相应的处理器上运行。一个任务τ_i所有子任务的执行时间总和等于C_i。例如在图 6.1 中，一个任务τ_i被切分成三个子任务τ_i^1，τ_i^2和τ_i^3，并分别被分配到处理器P_1，P_2和P_3上。

为了使一个切分任务能够正确的串行执行，其各个子任务需要相互同步。例如在图 2.1 中，在同一次任务释放的工作中，子任务τ_i^2不能在子任务τ_i^1完成前开始执行。这相当于将τ_i^2的实际就绪时间相对于τ_i本身的释放时间延迟（最多）R_i^1个时间单位，这里R_i^1是子任务τ_i^1的最坏响应时间上限。这还可以被看作是把τ_i^2的相对截止期（相对于τ_i原来的相对截止期）缩短（最多）R_i^1个时间单位。类似地，子任务τ_i^3的实际就绪时间被延迟（最多）$R_i^1+R_i^2$个时间单位，即τ_i^3的实际相对截止期被缩短了（最多）$R_i^1+R_i^2$个时间单位。本章使用τ_i^k来表示一个切分任务τ_i的第k个子任务，并定义τ_i^k的实际相对截止期为：

$$\Delta_i^k = T - \sum_{l\in[1,k-1]} R_i^l$$

于是，可以用一个三元组$\langle c_i^k, T_i, \Delta_i^k \rangle$来代表一个子任务$\tau_i^k$，其中$c_i^k$是$\tau_i^k$的最坏情况执行时间，$T_i$是$\tau_i^k$的原始周期，$\Delta_i^k$是$\tau_i^k$的虚拟相对截止期。为了保持一致性，对于每一个非切分任务也可以用一个单独的子任务τ_i^1来表示，其中$c_i^1 = C_i$且$\Delta_i^1 = T_i$。

一个子任务τ_i^k的普通利用率定义为

$$U_i^k = \frac{c_i^k}{T}$$

除此之外，还为每个子任务定义一个新的利用率指标，虚拟利用率V_i^k，来描述子任务τ_i^k相对于实际相对截止期Δ_i^k的工作负载：

$$V_i^k = \frac{c_i^k}{\Delta_i^k}$$

本章将一个任务τ_i的最后一个子任务称为后部子任务，并用τ_i^t来表示。称其它子任务为前部子任务，并用τ_i^{bj}来表示任务τ_i的第 j 个前部子任务。用符号$\tau_i \mapsto P_q$来表示任务τ_i被分配到了处理器P_q上，并称P_q为任务τ_i宿主处理器。

一个任务集τ在一个准划分调度算法A下是可调度的，当且仅当在根据A的划分算法将任务集分配到各个处理器上以后，运行时根据 A 的调度算法，任务集中的每一个任务都满足其截止期。

6.2 针对轻型任务集的算法 RMTS-1

本章提出的准划分调度算法 RMTS-1（Rate Monotonic with Task Splitting 1）与现有的其他准划分调度算法一个非常重要的区别在于，RMTS-1 的划分算法采用“最坏适用”（worst-fit）的分配原则，而其他现有算法的划分采用“最先适用”（first-fit）的分配原则[50, 52, 54]。

“最先适用”的分配原则的基本过程如下：首先选择一个处理器，在该处理器上分配进可能多的任务来填满它的处理能力，然后再选择下一个处理器，并重复上述过程。相反地，“最坏适用”的分配原则总是选择所有处理器中目前已经被分配的任务利用率总和最少的那个来分配当前的任务。也就是说，“最坏适用”的分配原则下，各个处理器被占用的处理能力是交替增长的。

本章提出的算法 RMTS-1 采用“最坏适用”的分配原则的原因如下：一个子任务τ_i^k的实际相对截止期Δ_i^k要比其原始相对截止期T_i短，因而一个任务τ_i所有子任务的虚拟利用率之和高于该任务的原始利用率U_i。这个现象正是使准划分调度算法来达到与单处理机上相同的利用率界限的难点之所在。在“最坏适用”的分配原则之下，各个处理器被占用的处理能力是交替增长的，且仅在一个处理器的处理能力被全部占满以后才会发生任务的切分。因此，如果按照任务的优先级顺序来进行分配，那么在“最坏适用”的分配原则之下，切分任务通常具有比较高的优先级（与相应的处理器上的其它任务相比）。这个结果有利于使整个系统可调度，因为一个高优先级任务更容易满足截止期，也更加能够容忍对其相对截止期进行缩减的操作。考虑一个极端的情况：如果能够保证所有切分任务的所有子任务都在相应的处理器上具有最高优先级，那么就根本不需要去考虑缩短相对截止期对这些切分任务的子任务造成的影响。这是因为，既然这些子任务都具有各个相应的处理器上的最高优先级，那么它们一定是可以调度的。当这些切分任务的子任务被保证以后，就可以轻易地达到希望得到的Liu&Layland资源利用率界限。因此，本章提出的准划分调度算法背后的设计思想可以概括为：让切分任务的子任务在各个相应的处理器上拥有尽可能高的优先级。

相反地，在“最先适用” 分配原则下，一个切分任务的子任务有可能在其相应的宿主处理器上具有很低的优先级。例如，文献[54]中的准划分调度算法可以达到 65%的资源利用率界限，在这个算法中，最坏情况下每个切分任务的第二个子任务总是具有其宿主处理器上的最低优先级。

正如将要在下面的章节中阐明的，RMTS-1 并不能完全地解决问题以达到期望的资源利用率界限。更准确地说，RMTS-1 仅能对轻型任务系统达到Liu&Layland资源利用

率界限，其中轻型任务系统定义为每个任务的利用率都不超过$\frac{\Theta(N)}{1+\Theta(N)}$的任务系统。直观上说，这是因为如果一个任务的利用率非常高，即便使用最坏适用”的分配原则，改任务的后部子任务可能会在相应的处理器上具有比较低的优先级。在下一节中提出的第二个准划分调度算法 RMTS-2 将解决这个问题。

下面将要介绍算法 RMTS-1 以及其资源利用率界限性质。本章的余下部分是这样进行组织的：

1. 介绍 RMTS-1 的划分算法，
2. 证明任意一个满足$U(\tau)\le\Theta(N)$的任务集τ都能够成功地被 RMTS-1 的划分算法的所划分。
3. 介绍 RMTS-1 的调度算法。
4. 证明任意一个能够被 RMTS-1 的划分算法成功划分的轻型任务集在 RMTS-1 的调度算法下都是可调度的。

综合以上步骤，可以得出任意一个满足$U(\tau)\le\Theta(N)$的轻型任务集在 RMTS-1 下都是可调度的，并最终得出对于轻型任务集 RMTS-1 的资源利用率界限为$\Theta(N)$。

6.2.1 RMTS-1 的划分算法与调度算法

RMTS-1 的划分算法非常简单，其可以概括为以下规则：

- 根据从低到高的优先级顺序分配任务，并且每一步总是选择到目前为止所有处理器中已经被分配的任务的资源利用率总和最小的处理器来分配当前任务。
- 当一个任务（子任务）不能被整个地分配到当前所选的处理器上时，将其切分为两部分，使得把第一部分加入当前处理器后其利用率之和为$\Theta(N)$，然后把第二部分留到下一步中进行分配。

图 6.2 中是 RMTS-1 划分算法的准确描述。在算法中，UQ是一个保存所有还未进行分配的任务（子任务）的列表，其中的任务（子任务）按优先级从低到高的顺序排列。UQ初始化为$\{\tau_N^1,\tau_{N-1}^1,...,\tau_1^1\}$，其中每个元素$\tau_i^1=\langle c_i^1=C_i,T_i,\Delta_k^1=T_i\rangle$是任务$\tau_i$的子任务形式。数组$\Psi[1...M]$中的每个元素$\Psi[q]$代表处理器$P_q$上已经分配的任务（子任务）资源利用率的总和。

1: **if** $U(\tau)>\Theta(N)$ **then abort**
2: $UQ := [\tau_N^1, \tau_{N-1}^1, \cdots, \tau_1^1]$
3: $\Psi[1\cdots M] :=$ all zeros
4: **while** $UQ \neq \varnothing$ **do**
5: $P_q :=$ the processor with the minimal Ψ
6: $\tau_i^k := \text{pop\_front}(UQ)$
7: **if** $(U_i^k + \Psi[q] \leq \Theta(N))$ **then**
8: $\tau_i^k \mapsto P_q$
9: $\Psi[q] := \Psi[q] + U_i^k$
10: **else**
11: split τ_i^k into two parts τ_i^k and τ_i^{k+1} such that $U_i^k + \Psi[q] = \Theta(N)$
12: $\tau_i^k \mapsto P$
13: $\Psi[q] := \Theta(N)$
14: $\text{push\_front}(\tau_i^{k+1}, UQ)$
15: **end if**
16: **end while**

图 6.2 RMTS-1 的划分算法伪代码

Fig. 6.2 Pseduo code of the partitioning algorithm of RMTS-1

RMTS-1 划分算法的工作流程如下。在每次循环里，将目前为止所有处理器中已经被分配的任务的资源利用率总和最小的处理器，记为 P_q。然后选择当前选择 UQ 中的第一个任务（子任务），记为 τ_i^k，进行分配。此时 τ_i^k 是所有还未被分配的任务中优先级最低的。首先尝试将 τ_i^k 整个分配到处理器 P_q 上。如果满足条件

$$U_i^k + \Psi[q] \leq \Theta(N)$$

则可以将 τ_i^k 整个分配到 P_q 上。否则，将 τ_i^k 切分为两个子任务 τ_i^k 和 τ_i^{k+1}，使其满足条件：

$$U_i^k + \Psi[q] = \Theta(N)$$

其中 $U_i^k = c_i^k / T_i$ 是切分后的第一个子任务 τ_i^k 的资源利用率。然后设置 $\Psi[q] := \Theta(N)$，这意味着 P_q 已经被分满了，之后不会再被分配任何其它子任务。最后将切分所得的第二个子任务 τ_i^{k+1} 放回 UQ 的第一个位置，以便在下一个循环里对其进行分配。这个过程重复直到所有的任务都被分配完。

根据 RMTS-1 划分算法的过程，可以知道任意一个在 Liu&Layland 资源利用率界限之下的任务集都能够被 RMTS-1 的划分算法成功划分，如以下引理所述。

引理 6.1：任意满足下述条件的任务集

$$U(\tau) \le \Theta(N) \tag{6.2}$$

都能够被 RMTS-1 的划分算法成功划分。

需要注意的是，RMTS-1 的划分算法不对系统的可调度性提供任何保障。系统的可调度性将在后面的章节被证明。

下面介绍 RMTS-1 的调度算法。运行时，每个处理器上的任务（子任务）根据单调速率算法（RMS）进行本地调度，即根据其原始周期来分配优先级（最短周期任务拥有最高优先级）。一个切分任务的各个子任务之间要遵循它们的先后顺序，即一个子任务 τ_i^k 在其前继子任务 τ_i^{k-1}（在另一个处理器上）执行结束以后才可以执行。

6.2.2 可调度性分析

本小节将要证明任意一个能够被 RMTS-1 的划分算法成功划分的轻型任务集在 RMTS-1 的调度算法下都是可调度的。首先介绍一个 RMTS-1 的重要性质：

引理 6.2：在进行 RMTS-1 划分算法之后，每个切分任务的每个前部子任务在其宿主处理器上具有最高优先级。

证明：在 RMTS-1 的划分算法中，任务切分仅在某个处理器的处理能力被完全占满时在发生。所以在一个前部子任务被分配到一个处理器上以后，将不再有其它（子）任务被分配到这个处理器上。由于任务的分配是按照优先级从低到高的顺序进行的，所以所有已经分配到这个处理器上的（子）任务的优先级都比这个前部子任务的优先级低。综上所述，每个前部子任务在其宿主处理器上具有最高优先级。 □

根据上述引理可知，每个前部子任务的响应时间等于其最坏情况执行时间，因此每个后部子任务的实际相对截止期 Δ_i^k 可以计算如下：

$$\Delta_i^t = T_i - \sum_{j\in[1,B]} c_i^{bj} = T_i - \left(C_i - c_i^t\right) \tag{6.3}$$

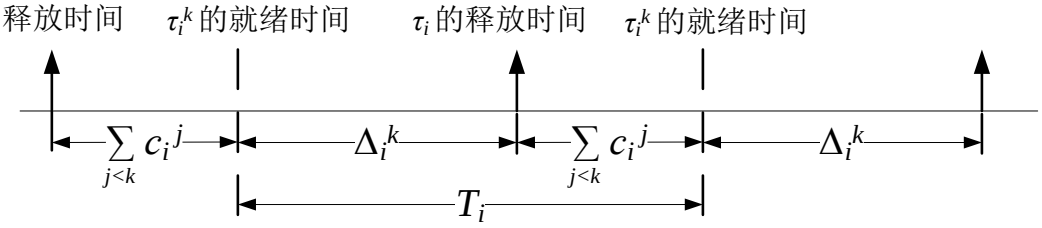


图 6.3 将每个子任务看作周期任务示意图

Fig. 6.3 Illustration of viewing each subtask as a periodic task

所以 RMTS-1 中每个处理器上的调度算法可以看作成这样的情况：不需要考虑一个

切分任务的各个子任务之间的同步，而是把每一个子任务τ_i^k看做一个独立的任务，这个独立的任务的周期等于其所属任务的原始周期T_i，但是其具有一个比较短的实际相对截止期Δ_i^k（根据公式(6.3)计算），如图 6.3 所示。

下面将要分别证明非切分任务的可调度性，切分任务的前部子任务的可调度性，已及切分任务的后部子任务的可调度性。

(1) 非切分任务与前部子任务的可调度性

引理 6.3：使用 RMTS-1 调度一个满足条件$U(\tau) \le \Theta(N)$的任务集，则每个非切分任务都能满足其截止期。

证明：首先，在 RMTS-1 中，分配到每个处理器的任务根据 RMS 进行调度，并且每个处理器上所分配的任务的资源利用率总和不超过$\Theta(N)$。其次，每个非切分任务的相对截止期依旧等于其原始周期而未发生改变，所以每个非切分任务都是可以调度的。需要注意的是，尽管其他切分任务的子任务的实际相对截止期变短了，但是这并不影响非切分任务的可调度性。这是因为只有子任务的周期，而非相对截止期对非切分任务的可调度性有影响。 □

引理 6.4：使用 RMTS-1 调度一个满足条件$U(\tau) \le \Theta(N)$的任务集，则每个前部子任务都能满足其截止期。

证明：每个前部子任务在其宿主处理器上具有最高优先级，因此一定可以满足其截止期。需要注意的是，通过递归，容易证明每个前部子任务的实际相对截止期一定不小于其最坏情况执行时间。 □

(2) 非切分任务与前部子任务的可调度性

现在来证明任意一个后部子任务的可调度性。假设任务τ_i被切分成 B 个前部子任务和一个后部子任务。使用$\tau_i^{bj}, j \in [1, B]$来表示$\tau_i$的第 j 个前部子任务，$\tau_i^t$表示$\tau_i$的后部子任务。用$U_i^{bj} = c_i^{bj} / T_i$和$U_i^t = c_i^t / T_i$表示$\tau_i^{bj}$和$\tau_i^t$的原始资源利用率。此外，定义下列概念（参见图 6.4）：

- 对应每个前部子任务τ_i^{bj}，定义X^{bj}为τ_i^{bj}的宿主处理器P_{bj}上所有优先级比τ_i^{bj}低的（子）任务的资源利用率之和。
- 对应每个后部子任务τ_i^t，定义X^t为τ_i^t的宿主处理器P_t上所有上所有优先级比τ_i^t低的（子）任务的资源利用率之和。
- 对应每个后部子任务τ_i^t，定义Y^t为τ_i^t的宿主处理器P_t上所有上所有优先级比τ_i^t高的（子）任务的资源利用率之和。

引理 6.5：假设一个后部子任务τ_i^t被分配到处理器P_t上。如果τ_i^t满足条件

$$Y^t \cdot T_i / \Delta_i^t + V_i^t \le \Theta(N) \tag{6.4}$$

则 τ_i^t 必能满足其截止期。

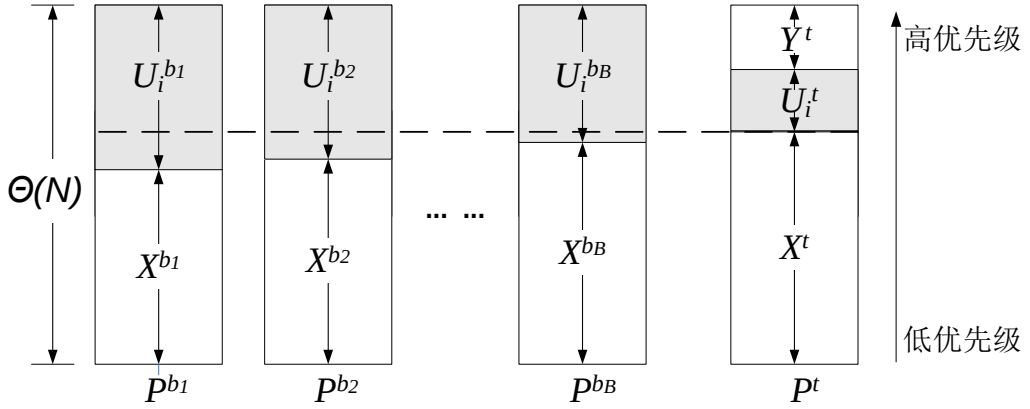


图 6.4 X^{bj}， X^t 和 Y^t 示意图

Fig. 6.4 Illustration of X^{bj}, X^t and Y^t

证明：证明思路如下：考虑任务集 Γ，其中含有后部子任务 τ_i^t 和所有同一处理器上的比优先级高的（子）任务，即那些资源利用率被包括在 Y^t 中的（子）任务。对于这个任务集，构造一个新的任务集 Γ：Γ 中的（子）任务与 Γ 中的（子）任务一一对应，但是对于 Γ 中每个周期比 Δ_i^t 大的任务，其 Γ 中的对应任务的周期缩短至 Δ_i^t。然后首选证明 Γ 中与 τ_i^t 相对应的（子）任务都可调度，再证明这意味着 Γ 中的 τ_i^t 也是可调度的。

下面是详细证明过程。令 P_t 为 τ_i^t 的宿主处理器。定义 Γ 为：

$$\Gamma = \left\{ \tau_h^k \mid \tau_h^k \mapsto P_t \wedge h \le i \right\} \tag{6.5}$$

现在来构造 Γ：对于每个（子）任务 $\tau_h^k \in \Gamma$，Γ 中有一个一一对应的（子）任务 τ_h^k。τ_h^k 和 τ_h^k 的唯一区别在于 τ_h^k 的周期可能根据如下规则被缩短：

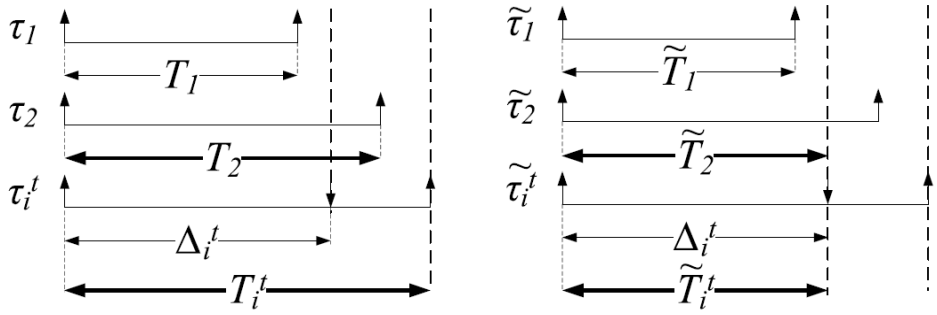
$$c_h^k = c_h^k, T_h = \begin{cases} T_h, \ if \ \ T_h \le \Delta_i^t \\ \Delta_i^t, \ if \ \ T_h > \Delta_i^t \end{cases}$$

Γ 中任务的优先级顺序与 Γ 中相应任务的优先级顺序一致，所以可知 Γ 中任务的优先级依然符合单调速率（RMS）顺序，即任务周期越小，优先级越高。

图 6.5 示例 Γ 的构造。在图 6.5(a)中，Γ 中含有三个（子）任务。其中 τ_1 的周期小于 Δ_i^t，而 τ_2 得周期大于 Δ_i^t。根据上述构造规则，如图 6.5(b)所示 Γ 也有三个（子）任务：τ_1，τ_2 and τ_i^t，其中 τ_2 和 τ_i^t 的周期被缩短至 Δ_i^t。

现在来证明 Γ 中 τ_i^t 的可调度性。首先考虑 Γ 中所有（子）任务的资源利用率总和：

$$U\left(\Gamma\right) = \sum_{\tau_h^k \in \Gamma} c_h^k / T_h = \sum_{\tau_h^k \in \Gamma \setminus \left\{\tau_i^t\right\}} c_h^k / T_h + V_i^k \tag{6.6}$$



(a) Γ (b) Γ

图 6.5 Γ 和 Γ 示意图

Fig. 6.5 Illustration of Γ and Γ

对于 Γ 中的一个（子）任务 τ_h^k，根据其周期是否被缩短了（即是否具有 $T_h \le \Delta_i^t$）而进行分类讨论：

- 如果 $T_h \le \Delta_i^t$，则可知 $T_h = T_h$。因为 $T_i > \Delta_i^t$，可知

$$c_h^k / T_h = c_h^k / T_h = U_h^k < U_h^k \cdot T_i / \Delta_i^t$$

- 如果 $T_h > \Delta_i^t$，则可知 $T_h = \Delta_i^t$。根据单调速率优先级顺序又可知 $T_h \le T_i$，因此可知：

$$c_h^k / T_h = c_h^k / \Delta_i^t \le c_h^k / T_h \cdot T_i / \Delta_i^t$$

$$c_h^k / T_h = U_h^k \cdot T_i / \Delta_i^t$$

两种情况下都满足 $c_h^k / T_h \le U_h^k \cdot T_i / \Delta_i^t$。将其应用于(6.6)可以得到：

$$U(\Gamma) \le \sum_{\tau_h^k \in \Gamma \setminus \{\tau_i^t\}} U_h^k \cdot T_i / \Delta_i^t + V_i^k \tag{6.7}$$

根据 Y^t 的定义又可知 $Y^t = \sum_{\tau_h^k \in \Gamma \setminus \{\tau_i^t\}} U_h^k$，将其应用于上式可得：

$$U(\Gamma) \le Y^t \cdot T_i / \Delta_i^t + V_i^t$$

根据条件(7)可知 $U(\Gamma) \le \Theta(N)$. 因此，τ_i^k 是可调度的。值得注意的是，尽管 Γ 中可能存在其它实际相对截止期比周期短的后部子任务，这并不影响使用条件 $U(\Gamma) \le \Theta(N)$ 来保证单调速率调度算法下 τ_i^t 的可调度性。

最后用 τ_i^t 的可调度性来证明 τ_i^t 的可调度性。Γ 和 Γ 的唯一区别在于，Γ 中某些（子）任务的周期可能比其在 Γ 中相对于的（子）任务要大。因此 τ_i^t 所承受的 Γ 中高优先级任务的干涉，一定不大于 τ_i^t 所承受的 Γ 中高优先级任务的干涉。又因为 τ_i^t 与 τ_i^t 的相对截止期相同，可知如果 τ_i^t 是可调度的，那么 τ_i^t 一定也是可调度的。 □

下面将要证明条件(6.4)。根据引理 5 可知所考虑的后部子任务是可调度的。如这一小节开始时所述，条件(6.4)对于 RMTS-1 并非普遍成立的，而仅对所谓的轻型任务集成

立。轻型任务集的定义如下：

定义 6.1：一个任务 τ_i 如果满足下述条件则为一个轻型任务：

$$U_i \le \frac{\Theta(N)}{1+\Theta(N)}$$

否则，其为一个重型任务。如果一个任务集 τ 中所有的任务都为轻型任务，则称其为一个轻型任务集。

引理 6.6：假设一个后部子任务 τ_i^t 被分配到处理器 P_t 上，如果 τ_i 是一个轻型任务，则下述条件一定成立：

$$Y^t \cdot T_i / \Delta_i^t + V_i^t \le \Theta(N)$$

证明：证明首先根据 X^{bj} 和 X^t 的性质，以及各个子任务间资源利用率的关系为 Y^t 找到一个上限。然后根据 Y^t 的上限以及 τ_i 是一个轻型任务这一性质，来证明引理成立。

每当一个（子）任务被切分成一个新的前部子任务和余下部分时，这个新的前部子任务的宿主处理器的资源利用率将变为 $\Theta(N)$。此外，每个前部子任务都具有其宿主处理器上的最高优先级，因此可知：

$$\forall j \in [1,B] : U_i^{bj} + X^{bj} = \Theta(N)$$

将所有 B 个这样的等式加在一起可得：

$$\sum_{j\in[1,B]} U_i^{bj} + \sum_{j\in[1,B]} X^{bj} = B \cdot \Theta(N) \tag{6.8}$$

现在考虑 τ_i^t 的宿主处理器，记为 P_t。P_t 上所有（子）任务的资源利用率总和为 $X^t + U_i^t + Y^t$，且知：

$$X^t + U_i^t + Y^t \le \Theta(N).$$

将上式应用于(6.8)可得：

$$Y^t \le \frac{\sum_{j\in[1,B]} U_i^{bj}}{B} + \frac{\sum_{j\in[1,B]} X^{bj}}{B} - U_i^t - X^t \tag{6.9}$$

在划分阶段，每一步中总是选择所以处理器中已经被分配的（子）任务的资源利用率总和最小处理器来分配当前任务。据此可知对于每一个 τ_i^{bj} 都有 $X^{bj} \le X^t$。因此所有的 X^{bj} 之和不大于 $B \cdot X^t$。因此可以将(6.9)化为：

$$Y^t \le \frac{\sum_{j\in[1,B]} U_i^{bj}}{B} - U_i^t$$

又因为 $B \ge 1$ 和 $\sum_{j\in[1,B]} U_i^{bj} = U_i - U_i^t$，上式可以化为：

$$Y^t \le U_i - 2 \cdot U_i^t$$

至此，已经为Y^t找到了一个上限$U_i-2\cdot U_i^t$。根据此上限可以得到：

$$Y^t\cdot\frac{T_i}{\Delta_i^t}+V_i^t\le\left(U_i-2\cdot U_i^t\right)\cdot\frac{T_i}{\Delta_i^t}+V_i^t \tag{6.10}$$

剩余工作是证明上式中不等式的右边项不会超过$\Theta(N)$。证明的关键是将其化为一个适合使用τ_i是一个轻型任务这一性质的形式。

首先因为τ_i^t的实际相对截止期是由其原始周期T_i减掉τ_i所有前部子任务的执行时间所得，即$\Delta_i^t=T_i-\left(C_i-c_i^t\right)$。此外，还知道$U_i=C_i/T_i$，$U_i^t=c_i^t/T_i$以及$V_i^t=c_i^t/\Delta_i^t$。综合这些关系可得：

$$\left(U_i-2\cdot U_i^t\right)\cdot\frac{T_i}{\Delta_i^t}+V_i^t=\frac{C_i-c_i^t}{T_i-\left(C_i-c_i^t\right)}$$

$$\left(U_i-2\cdot U_i^t\right)\cdot\frac{T_i}{\Delta_i^t}+V_i^t=\frac{T_i}{T_i-\left(C_i-c_i^t\right)}-1$$

因为$c_i^t>0$，可以为上式的右边项找到一个上限：

$$\frac{T_i}{T_i-\left(C_i-c_i^t\right)}-1<\frac{T_i}{T_i-C_i}-1 \tag{6.11}$$

因为τ_i是一个轻型任务，可知

$$U_i\le\frac{\Theta(N)}{1+\Theta(N)}$$

将$U_i=C_i/T_i$应用于上式可得：

$$\frac{T_i}{T_i-C_i}-1\le\Theta(N)$$

将上式与(6.10)和(6.11)相结合，最终得到$Y^t\cdot T_i/\Delta_i^t+V_i^t\le\Theta(N)$。 □

根据引理6.5与引理6.6，可以立即证明后部子任务的可调度性。

引理 6.7：使用RMTS-1调度一个满足条件$U(\tau)\le\Theta(N)$的任务集，则每个后部子任务都能满足其截止期。

6.2.3 RMTS-1的资源利用率界限

根据引理 6.1 可知，如果一个任务集τ的资源利用率总和不超过$\Theta(N)$，则其必能被RMTS-1的划分算法成功划分。根据引理3和引理4可知，这样一个任务集在运行时，τ中所有的非切分任务与前部子任务都能满足截止期。根据引理 6.7 可知，如果τ中一个轻型任务τ_i被切分，那么τ_i的后部子任务τ_i^t必能满足截止期。在一般情况下无法在进行系统的划分之前就知道哪个任务将要被切分，因此需将轻型任务这一个约束置于τ中

所有的任务，来得到一个充分的可调度性分析条件：

定理 6.2：令τ为一个轻型任务集。如果下述条件被满足则τ在 M 个处理器上可以被 RMTS-1 调度：

$$U(\tau) \le \Theta(N) \tag{6.12}$$

换言之，对于任意只含有资源利用率不超过$\Theta(N)/(1+\Theta(N))$的任务的任务集，RMTS-1 的资源利用率界限为$\Theta(N)$。

$\Theta(N)$是关于N的递减函数，所以这个资源利用率界限对于含有较少任务的任务集会变得更高。用N^*来表示分配到每个处理器上的（子）任务的个数，因此$\Theta(N^*)$严格大于$\Theta(N)$。$\Theta(N^*)$也可以作为每个处理器上的资源利用率界限。因此，可以在上面的证明中用$\Theta(N^*)$替代所有的$\Theta(N^*)$，于是可以知道，对于任意一个只含有资源利用率不超过$\Theta(N^*)/(1+\Theta(N^*))$的任务的任务集，RMTS-1 的资源利用率界限为$\Theta(N^*)$。

容易看出每个处理器上至少分配了一个任务，且一个切分任务的两个子任务不能被分配到同一个处理器上。所以每个处理器上所分配的（子）任务的个数最多为$N-M+1$。因此可以令$N^* = N-M+1$来得到一个更准确的上限。

6.3 面向任意任务集的算法 RMTS-2

本节将介绍本章的第二个准划分固定优先级调度算法 RMTS-2。RMTS-2 对任意任务集的资源利用率界限都是$\Theta(N)$。

正如在第三节的开始讨论的那样，本章提出的算法能够达到较高的资源利用率界限的关键在于使每个切分任务在其宿主处理器上具有尽可能高的的优先级。在上一节介绍的第一个算法 RMTS-1 中，如果一个任务的资源利用率非常高，那么它的后部子任务有可能在其宿主处理器上有比较低的优先级。图 6.6 中给出了这样一个例子。这正是为什么第一个算法 RMTS-1 的资源利用率界限对于含有重型任务的任务集不成立。

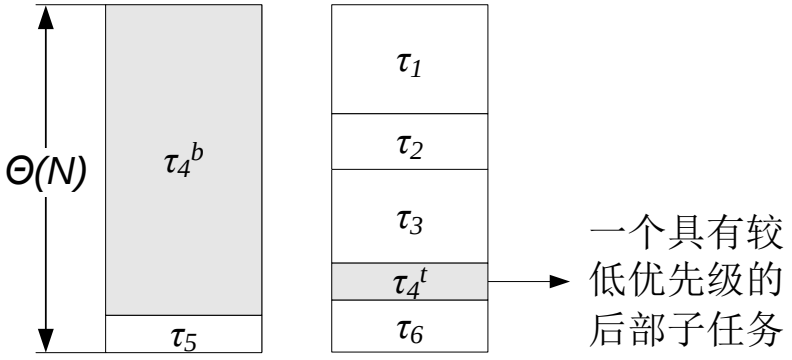


图 6.6 一个具有较低优先级的后部子任务示意图

Fig. 6.6 Illustration of a tail subtask with a low priority level

为了解决这个问题，本节提出第二个准划分算法 RMTS-2。RMTS-2 主要的特点在于：在进行整个任务系统的统一分配以前，先对那些有可能具有较低优先级后部子任务的重型任务进行预分配，以使的这些重型任务不会被切分。

任务	C_i	T_i	是否重型任务？	优先级
τ_1	3	4	是	高
τ_2	4.25	10	否	中
τ_3	4.25	10	否	低

图 6.7 一个例子任务集

Fig. 6.7 A task set example

需要注意的是，如果简单地预分配所有的重型任务，依然有可能使某些任务的后部子任务在其宿主处理器上具有比较低的优先级。考虑表 X 中的任务集，假设其运行于两个个处理器上。为了简单起见，假设 $\Theta(N)=0.8$ 和 $\Theta(N)/(1+\Theta(N))=4/9$。如果将重型任务 τ_1 进行预分配，然后再用 RMTS-1 分配余下的两个任务 τ_2 和 τ_3，则分配过程如下：

1. $\tau_1 \mapsto P_1$；
2. $\tau_3 \mapsto P_2$；
3. τ_2 不能够被整个地分配到 P_2 上，所以将其切分成两个子任务 $\tau_2^1=\langle 3.75,10,10\rangle$ 和 $\tau_2^2=\langle 0.5,10,6.25\rangle$，且 $\tau_2^1 \mapsto P_2$；
4. $\tau_2^2 \mapsto P_1$。

然后每个处理器上用 RMS 进行调度。可以看出 τ_2^2 在 P_1 具有最低优先级，并将错过其截止期。但是，如果不对 τ_1 进行预分配而直接使用 RMTS-1 进行分配，整个任务集则可调度。

为了解决这个问题，RMTS-2 中将使用一个更加灵活的预分配机制。直观上讲，RMTS-2 将精确地选择一部分重型任务进行预分配，而不会引起任何其它后部任务错失其截止期。这是通过一个简单的测试来完成的。而那些不能通过这个测试的重型任务将不会被预分配，而和其它轻型任务一起进行普通的分配。这个方法成功的关键在于，对那些不能通过测试的重型任务，可以证明它们的后部子任务也不会错失截止期。

6.3.1 RMTS-2 的划分算法与调度算法

首先介绍一些概念。如果一个重型任务 τ_i 被预分配到了一个处理器 P_q 上，则称 τ_i 为一个预分配任务，否则称其为一个普通任务；称 P_q 为一个预分配处理器，否则称其为一个普通处理器。

RMTS-2 的划分算法包含以下三个主要步骤：

1. 首先将满足某个特定条件的所有重型任务每个预分配到一个处理器上。
2. 将剩下的任务（即普通任务）用 RMTS-1 在剩下的处理器（即普通处理器）上进行分配，直到所有的普通处理器都被放满（子）任务。
3. 将剩下的任务在预分配处理器上进行分配。分配的方法为选择一个处理器然后分配尽可能多的（子）任务，直到其被放满，然后再选择下一个处理器继续进行分配。

算法的详细过程见图 6.8。首先介绍算法中使用的一些数据结构的含义：

- PQ 是一个处理器队列。其初始状态为$[P_1, P_2, \ldots, P_M]$。该队列的删除与插入操作均只从队列的头部进行。
- PQ_{pre} 是一个放置预分配处理器的队列。其出示状态为空。
- UQ 是一个放置算法步骤 1）之后未被分配任务的队列。其出示状态为空。在算法的步骤 1）中，每个被判定为将不被预分配的任务 τ_i 将以其子任务形式 τ_i^1 从头部放入队列 UQ。该队列的删除与插入操作均只从队列的头部进行。
- $\Psi[1...M]$ 数组与在算法 RMTS-1 中的意义相同：其中的每个元素 $\Psi[q]$ 表示处理器 P_q 上已经被分配的所有任务的资源利用率的总和。

任务	C_i	T_i	是否重型任务?	优先级
τ_1	0.5	10	否	最高
τ_2	4.5	10	是	
τ_3	6	10	是	
τ_4	4	10	否	
τ_5	3	10	否	
τ_6	6	10	是	
τ_7	3	10	否	最低

图 6.8 一个示例 RMTS-2 的例子任务集

Fig. 6.8 A task set example demonstrating RMTS-2

下面将用图 6.8 中的任务集和一个四处理器的平台来实例算法 RMTS-2 的工作过程。为了简单起见，假设 $\Theta(N)=0.7$，进而轻型任务的资源利用率上限 $\Theta(N)/(1+\Theta(N))$ 约等于 0.41。算法开始时，上述数据结构的初始状态为：

- $PQ=[P_1, P_2, P_3, P_4]$
- $PQ_{pre}=\varnothing$
- $UQ=\varnothing$
- $\Psi[1...4]=[0,0,0,0]$

算法的步骤 1）中（6 至 15 行），τ 中的每个任务根据优先级从高到低的顺序依次

被访问。如果τ_i为重型任务，则对其考查下述条件：

$$\sum_{j>i} U_j \le (|PQ|-1)\cdot\Theta(N) \tag{6.13}$$

其中$|PQ|$是到目前为止PQ中所剩的处理器的个数。如果一个重型任务满足τ_i上述条件，则决定将其预分配到一个处理器上。上述条件的直观意义是：如果预分配任务τ_i，则在剩余的处理器上一定有足够的处理能力来接受剩下的任务中优先级较低的任务。这样一来，优先级较低的后部子任务将不会被分配到τ_i的宿主处理器上了。

在所用的例子中，首先访问第一个任务τ_1^1。τ_1^1是一个轻型任务，所以将其放入UQ中。下一个任务τ_2是重型任务，此时$|PQ|=4$，所以条件(6.13)不被满足，所以将不会预分配τ_2，并把它放入UQ的头部。下一个任务τ_3是重型任务，此时$|PQ|=4$，条件(6.13)被满足，因此将τ_3预分配到P_1上，并把P_1放入PQ_{pre}的头部（行 8 至 10）。τ_4和τ_5均为轻型任务，所以把它们先后放入UQ。下一个任务τ_6是重型任务，此时$|PQ|=3$（P_1已经被拿出PQ并放到PQ_{pre}里），条件(6.13)被满足，所以预分配τ_5到P_2，并把P_2放到PQ_{pre}的头部。最后一个任务τ_7为轻型任务，所以把它放到UQ的头部。至此，算法的步骤 1）完成，各个数据结构的状态如下：

- $PQ=[P_3,P_4]$
- $PQ_{pre}=[P_2,P_1]$
- $UQ=[\tau_7^1,\tau_5^1,\tau_4^1,\tau_2^1,\tau_1^1]$
- $\Psi[1...4]=[0.6,0.6,0,0]$

值得注意的是，PQ_{pre}中的处理器是根据其上的预处理任务的优先级从高到低排列的，UQ中的任务是根据优先级从高到低的顺序排列的。

算法的步骤 2）与 3）都在第 16 至第 35 行之间的循环内。步骤 2）中，剩下的任务（现都在UQ中）将被分配到普通处理器上（PQ中的处理器）。只有当PQ中所有的处理器都被放满了以后，才进入步骤 3），来把任务分配到PQ_{pre}中的处理器上。

在步骤 2）中分配一个任务τ_i^k的操作与 RMTS-1 相同。如果τ_i^k不需要切分就能被整个地分配到处理器P_q上，则$\tau_i^k \mapsto P_q$并更新$\Psi[q]$（行 24 至 28）。如果P_q是一个预分配处理器，则将其放回PQ_{pre}的头部（行 26 至 28），以使其在下一轮中继续被选择，否则不需要进行放回操作。

如果τ_i^k不能整个被分配到P_q上，则τ_i^k被切分成一个新的τ_i^k和另一个子任务τ_i^{k+1}，使得P_q在被分配τ_i^k以后的所有任务资源利用率之和为$\Theta(N)$，并将τ_i^{k+1}放回UQ。

1: **if** $U(\tau) > \Theta(N)$ **then abort**
2: $PQ := [P_1, P_2, \cdots, P_M]$
3: $PQ_{pre} := \varnothing$
4: $UQ := \varnothing$
5: $\Psi[1 \cdots M] :=$ all zeros
6: **for** $i := 1$ to N **do**
7: **if** τ_i is heavy $\wedge$
$\sum_{j>i} U_j \le (|PQ| - 1) \cdot \Theta(N)$ **then**
8: $P_q := \text{pop\_front}(PQ)$
9: Pre-assign τ_i to P_q
10: $\text{push\_front}(P_q, PQ_{pre})$
11: $\Psi[q] := \Psi[q] + U_i$
12: **else**
13: $\text{push\_front}(\tau_i^1, UQ)$
14: **end if**
15: **end for**
16: **while** $UQ \neq \varnothing$ **do**
17: $\tau_i^k := \text{pop\_front}(UQ)$
18: **if** $\exists P_q \in PQ : \Psi[q] \neq \Theta(N)$ **then**
19: $P_q :=$ the element in PQ with the minimal Ψ
20: **else**
21: $P_q := \text{pop\_front}(PQ_{pre})$
22: **end if**
23: **if** $U_i^k + \Psi[q] \le \Theta(N)$ **then**
24: $\tau_i^k \mapsto P_q$
25: $\Psi[q] := \Psi[q] + U_i^k$
26: **if** P_q came from PQ_{pre} **then**
27: $\text{push\_front}(P_q, PQ_{pre})$
28: **end if**
29: **else**
30: split τ_i^k into two parts τ_i^k and τ_i^{k+1} such that
$U_i^k + \Psi[q] = \Theta(N)$
31: $\tau_i^k \mapsto P_q$
32: $\Psi[q] = \Theta(N)$
33: $\text{push\_front}(\tau_i^{k+1}, UQ)$
34: **end if**
35: **end while**

图 6.9 RMTS-2 的划分算法伪代码

Fig. 6.9 Pseduo code of the partitioning algorithm of RMTS-2

值得注意的是，在向普通处理器和向预分配处理器上分配任务有一个重要的区别。当向普通处理器上分配任务时，算法总是选择具有最小Ψ元素值的处理器（这与RMTS-1相同）。不同的是，在向预分配处理器上分配任务的时候，总是选择PQ_{pre}的第一个处理器，即算法总是选择PQ_{pre}中拥有最低优先级预处理任务的处理器，并向其上分配尽可能多的（子）任务，直到它被放满为止。正如将要在下面的可调度性证明中看到的，这个特定的选择预分配处理器的顺序，与条件(6.13)一起，将是保证重型任务可调度性的关键。

在使用的例子中，所有剩下的任务将首先通过和RMTS-1同样的方式被分配到处理器P_3和P_4上。因此，进行如下分配：$\tau_7^1 \mapsto P_3$，$\tau_5^1 \mapsto P_4$，$\tau_4^1 \mapsto P_3$，然后将τ_2^1切分成$\tau_2^1 = \langle 4,10,1 \rangle$和$\tau_2^2 = \langle 0.5,10,6 \rangle$，然后$\tau_2^1 \mapsto P_4$。至此，所以的普通处理器都被分配满了，各个数据结构的状态如下：

- $PQ = [P_3, P_4]$
- $PQ_{pre} = [P_2, P_1]$
- $UQ = [\tau_2^2, \tau_1^1]$
- $\Psi[1...4] = [0.6, 0.6, 0.7, 0.7]$

接下来UQ中剩余的任务将被分配到预分配处理器上。首先，进行分配$\tau_2^2 \mapsto P_2$。

这之后P_2还没有被分满，所以依然在PQ_{pre}的头部。所以下一个（子）任务τ_1^1依然被分配到P_2上。至此，所有的任务都已经分配完毕，算法结束。

根据RMTS-2的算法过程，可以容易得到以下引理：

引理 6.8：任何一个满足下述条件的任务集

$$U(\tau) \le \Theta(N)$$

都可以被RMTS-2成功地划分。

上面介绍了RMTS-2的划分过程，现在来介绍RMTS-2的调度算法。RMTS-2的调度算法与 RMTS-1 相同：运行时，每个处理器上的任务（子任务）根据单调速率算法（RMS）进行本地调度，即根据其原始周期来分配优先级（最短周期任务拥有最高优先级）。一个切分任务的各个子任务之间要遵循它们的先后顺序，即一个子任务τ_i^k在其前继子任务τ_i^{k-1}（在另一个处理器上）执行结束以后才可以执行。

与RMTS-1一样，在RMTS-2中每个前部子任务都在其宿主处理器上具有最高的优先级。这点将在下面的小节中证明。与RMTS-1一样，在RMTS-2中每个处理器上的任务可以被看作一个独立任务集，其中每个子任务的实际相对截止期被缩短了，其缩短幅度为这个子任务所有前继子任务的最坏情况执行时间之和。

6.3.2 RMTS-2的性质

这一小节中将证明一些RMTS-2的重要性质。

引理 6.9：令τ_i为一个重型任务，且有η个预分配任务的优先级高于τ_i，则可知：如果τ_i是预分配任务，则其必满足

$$\sum_{j>i} U_j \le (M-\eta-1)\cdot\Theta(N) \tag{6.14}$$

如果τ_i不是预分配任务，则其必满足

$$\sum_{j>i} U_j > (M-\eta-1)\cdot\Theta(N) \tag{6.15}$$

证明：根据RMTS-2的划分算法过程可证。 □

引理 6.10：每个预分配任务在其宿主处理器上具有最低优先级。

证明：不失一般性，将所有的处理器按照如下方式顺序放在队列Q中：首先将所有的预分配处理器按照其上的预分配任务的优先级从高到低的顺序放在Q中，然后再后面以任意顺序放入所有的普通处理器。用P_x表示Q中的第x个处理器。假设τ_i是P_q上的预分配任务。

因为τ_i是一个预分配任务，且比τ_i优先级高的预分配任务的个数为$q-1$，所以根据引理6.9可知：

$$\sum_{j>i} U_j \le (M-q)\cdot\Theta(N) \tag{6.16}$$

在RMTS-2的划分算法中，只有当所有的普通处理器都被分配慢以后，普通任务才会被分配到预分配处理器上。而且预分配处理器是根据其上分配的预分配任务优先级从低到高的顺序选择的，所以可知只有当处理器$P_{q+1}...P_M$都被分配满了以后，普通任务才可能被分配到P_q上。处理器$P_{q+1}...P_M$上总共能提供的资源为$(M-q)\cdot\Theta(N)$（在本章提出的算法中当一个处理器上被占用的资源利用率为$\Theta(N)$时则称为被分满），并且根据（18）可知，当开始分配普通任务至P_q上时，所有比τ_i优先级低的普通任务都已经被分配到了处理器$P_{q+1}...P_M$上，所以所有被分配到P_q上的普通任务的优先级都比τ_i高。 □

引理 6.11：每个切分任务的前部子任务都在其宿主处理器上具有最高优先级。

证明：给定一个被分配到P_{bj}上的前部子任务τ_i^{bj}。因为任务的切分只在一个处理器被分满时发生，且所有普通任务是根据优先级从低到高的顺序分配，可知τ_i^{bj}在P_{bj}上所有的普通任务中具有最高优先级。此外，根据引理6.10可知如果P_{bj}是一个预分配处理器，则P_{bj}上的预分配任务的优先级也低于τ_i^{bj}，综上可知τ_i^{bj}的优先级为P_{bj}上最高。 □

6.3.3 RMTS-2 可调度性分析

根据引理 6.11 可知在 RMTS-2 中每个前部子任务具有其宿主处理器上最高优先级，因此可知所有的前部子任务都是可调度的。

RMTS-2 的调度算法是单调速率算法，并且每个非切分任务的相对截止期等于其周期，所以所以的非切分任务也都是可调度的。这可以通过与 RMTS-1 中引理 3 相同的方式来证明。

下面将要证明后部子任务的可调度性。假设任务 τ_i 被切分成 B 个前部子任务和一个后部子任务。与上一节中一样，使用 $\tau_i^{bj}, j \in [1, B]$ 来表示 τ_i 的第 j 个前部子任务，τ_i^t 表示 τ_i 的后部子任务。用 $U_i^{bj} = c_i^{bj} / T_i$ 和 $U_i^t = c_i^t / T_i$ 表示 τ_i^{bj} 和 τ_i^t 的原始资源利用率。对应每个前部子任务 τ_i^{bj}，定义 X^{bj} 为 τ_i^{bj} 的宿主处理器 P_{bj} 上所有优先级比 τ_i^{bj} 低的（子）任务的资源利用率之和。对应每个后部子任务 τ_i^t，定义 X^t 为 τ_i^t 的宿主处理器 P_t 上所有上所有优先级比 τ_i^t 低的（子）任务的资源利用率之和。对应每个后部子任务 τ_i^t，定义 Y^t 为 τ_i^t 的宿主处理器 P_t 上所有上所有优先级比 τ_i^t 高的（子）任务的资源利用率之和。

首先回顾上一节中的用来证明 RMTS-1 中后部子任务的引理 5：如果一个后部子任务 τ_i^t 满足下述条件：

$$Y^t \cdot T_i / \Delta_i^t + V_i^t \le \Theta(N) \tag{6.17}$$

则 τ_i^t 必能满足其截止期。这一结论对 RMTS-2 同样成立。这是因为在引理 6.5 的证明，唯一需要的性质是每个处理器上由单调速率算法进行调度，而 RMTS-2 上正是如此。因此，证明 RMTS-2 中的后部子任务的可调度性问题可以转化为证明在 RMTS-2 下所有的后部子任务都满足条件(6.17)。

在进行证明之前，先介绍一些概念。如果一个任务 τ_i 是一个重型任务，则称其后部子任务都重型后部子任务，否则称这个后部子任务为轻型后部子任务。下面将要分三种情况证明条件(6.17)：

1. τ_i^t 是一个轻型后部子任务，且其宿主处理器为普通处理器。
2. τ_i^t 是一个轻型后部子任务，且其宿主处理器为预分配处理器。
3. τ_i^t 是一个重型后部子任务。

以通过与 RMTS-1 相同的方式证明。这是因为 RMTS-2 在普通处理器上的划分与调度算法与 RMTS-1 都相同。实际上可以将 RMTS-2 中普通处理器上的划分与调度看作使用 RMTS-1 对一部分任务（那些被分配到普通处理器上的任务）在一部分处理器（普通处理器）上进行划分。因此在此情况下 τ_i^t 的可调度性可以通过与引理 6.6 相同的方式进行证明。

下面证明情况 2)：τ_i^t 是一个轻型后部子任务，且其宿主处理器 P_t 为预分配处理器。

引理 6.12：令 τ_i^t 为一个在 RMTS-2 中被分配到预分配处理器 P_t 上的轻型后部子任务，可知：

$$Y^t \cdot T_i / \Delta_i^t + V_i^t \le \Theta(N)$$

证明：根据引理 6.10 可知 τ_i^t 的优先级高于 P_t 上预分配任务的优先级，因此 X^t 大于或等于这个预分配任务的资源利用率。又因为一个预分配任务一定为重型任务，可知：

$$X^t > \frac{\Theta(N)}{1+\Theta(N)} \tag{6.18}$$

另一方面，因为 τ_i 为轻型任务，可知

$$\frac{C_i}{T_i} \le \frac{\Theta(N)}{1+\Theta(N)}$$

用 c_i^B 来表示 τ_i 所有前部子任务的执行时间之和。因为 $c_i^B < C_i$ 且 $\Theta(N)<1$，可知：

$$c_i^B < \frac{1}{1+\Theta(N)} \cdot T_i$$

$$\Leftrightarrow T_i\left(1-\frac{1}{1+\Theta(N)}\right) < T_i - c_i^B$$

$$\Leftrightarrow \frac{T_i}{T_i - c_i^B}\left(\Theta(N)-\frac{\Theta(N)}{1+\Theta(N)}\right) < \Theta(N)$$

$$\Leftrightarrow \frac{T_i}{\Delta_i^t}\left(\Theta(N)-\frac{\Theta(N)}{1+\Theta(N)}-U_i^t\right)+V_i^t < \Theta(N) \tag{6.19}$$

根据(6.18)和(6.19)可得

$$\frac{T_i}{\Delta_i^t}\left(\Theta(N)-X^t-U_i^t\right)+V_i^t < \Theta(N)$$

又因为每个处理器上的资源利用率之和不超过 $\Theta(N)$，即

$$Y^t \le \Theta(N)-X^t-U_i^t$$

最终可得 $Y^t \cdot T_i / \Delta_i^t + V_i^t < \Theta(N)$。 □

下面将证明情况 3)：τ_i^t 是一个重型后部子任务。在这个情况里，P_t 即可以是预分配处理器也可以是普通处理器。

引理 6.13：令 τ_i^t 是普通重型任务 τ_i 的后部子任务，则有：

$$Y^t \cdot T_i / \Delta_i^t + V_i^t \le \Theta(N)$$

证明：根据引理 6.9 中关于普通重型任务的性质可知 τ_i 满足条件

$$\sum_{j>i} U_j > (M-\eta-1)\cdot\Theta(N)$$

其中η为优先级比τ_i高的预分配任务的个数。

使用$\mathbb{M}$来表示所有处理器的集合（$|\mathbb{M}|=M$），并使用$\mathbb{H}$来表示所含预分配任务优先级高于τ_i的预分配处理器的集合，因此可知$|\mathbb{H}|=\eta$。根据上述定义可知

$$\sum_{j>i} U_j > (M-|\mathbb{H}|-1)\cdot\Theta(N) \tag{6.20}$$

根据引理 6.10 可知被分配到一个预分配处理器上的任意普通任务的优先级都高于这个处理器上的预分配任务的优先级，因此τ_i的前部子任务与后部子任务都被分配在了集合$\mathbb{M}\setminus\mathbb{H}$中的处理器上。因为在处理普通任务之前，预分配任务已经被全部分配完，且所有的普通任务是按照优先级由低至高的顺序进行分配，因此当划分算法开始分配τ_i时，所有比τ_i优先级低的任务都已经被分配到了集合$\mathbb{M}\setminus\mathbb{H}$中的处理器上。

使用$\mathbb{K}$来表示$\mathbb{M}\setminus\mathbb{H}$中那些不包含$\tau_i$的子任务的处理器的集合。对于每个$\mathbb{K}$中的处理器$P_k$，使用$X^k$来表示$P_k$上优先级低于$\tau_i$的所有任务的资源利用率之和。根据此定义可知：

$$X^t + \sum_{j\in[1,B]} X^{b_j} + \sum_{k\in[1,|\mathbb{K}|]} X^k = \sum_{j>i} U_j$$

定义可知$|\mathbb{K}|=M-|\mathbb{H}|-(B+1)$，又因为$X_k \le \Theta(N)$，可知：

$$X^t + \sum_{j\in[1,B]} X^{b_j} \ge \sum_{j>i} U_j - (M-|\mathbb{H}|-(B+1))\cdot\Theta(N) \tag{6.21}$$

根据不等式(6.20) 与(6.21)可以得到：

$$X^t + \sum_{j\in[1,B]} X^{bj} > B\cdot\Theta(N) \tag{6.22}$$

又知P_t上所有任务的资源利用率总和不超过$\Theta(N)$，即

$$Y^t \le \Theta(N) - X^t - U_i^t \tag{6.23}$$

根据不等式(6.22) 与 (6.23) 可得

$$Y^t \le \Theta(N) - \left(B\cdot\Theta(N) - \sum_{j\in[1,B]} X^{bj} \right) - U_i^t$$

又根据τ_i的资源利用率$U_i = U_i^t + \sum_{j\in[1,B]} U_i^{bj}$，可得

$$Y^t \le \Theta(N) - B\cdot\Theta(N) - U_i + \left(\sum_{j\in[1,B]} X^{bj} + \sum_{j\in[1,B]} U_i^{bj} \right) \tag{6.24}$$

因为每个前部子任务都在其宿主处理器上具有最高优先级，且每个含有前部子任务的处理器上的资源利用率的总和为$\Theta(N)$，可知

$$\sum_{l\in[1,B]} X^{bl} + \sum_{l\in[1,B]} U_i^{bl} = B\cdot\Theta(N) \tag{6.25}$$

根据(6.24)和(6.25)可得：

$$Y^t \le \Theta(N) - U_i$$
$$\Leftrightarrow Y^t\cdot T_i/\Delta_i^t + V_i^t \le \left(\Theta(N)-U_i\right)\cdot T_i/\Delta_i^t + V_i^t$$

将$U_i = C_i/T_i$ 和 $V_i^t = c_i^t/\Delta_i^t$ 应用于上式可得

$$Y^t\cdot T_i/\Delta_i^t + V_i^t \le \Theta(N)\cdot T_i/\Delta_i^t - C_i/\Delta_i^t + c_i^t/\Delta_i^t \tag{6.26}$$

用c_i^B来表示τ_i的所有前部子任务的执行时间之和，于是可知$c_i^t + c_i^B = C_i$和$\Delta_i^t = T_i - c_i^B$，并将其应用于不等式(6.26)，则可得

$$Y^t\cdot T_i/\Delta_i^t + V_i^t \le \frac{T_i\Theta(N) - c_i^B}{T_i - c_i^B} \tag{6.27}$$

因为 $\Theta(N)<1$，可知

$$c_i^B > c_i^B\cdot\Theta(N)$$
$$\Leftrightarrow T_i\cdot\Theta(N) - c_i^B < T_i\cdot\Theta(N) - c_i^B\cdot\Theta(N)$$
$$\Leftrightarrow \frac{T_i\cdot\Theta(N) - c_i^B}{T_i - c_i^B} < \Theta(N) \tag{6.28}$$

最终根据不等式(6.27)和(6.28)可得

$$Y^t\cdot T_i/\Delta_i^t + V_i^t < \Theta(N)$$

由此，定理可证。 □

6.4 小结

本章提出了两个准划分固定优先级调度算法来达到著名的Liu&Layland资源利用率界限。这是目前为止唯一能够将单处理机上的Liu&Layland资源利用率界限推广到多处理机调度的算法。本章提出的算法中分为划分具有线性复杂度，因此具有非常高的执行效率，这非常适合在复杂系统设计中的状态空间挖掘。但与此同时，本章提出的算法也有一定的缺点：其由于使用Liu&Layland资源利用率界限作为每个处理器上所能分配任务可调度性测试的唯一依据，因此在此算法下，每个处理器只能使用由Liu&Layland资源利用率界限所限制的系统资源，从系统评价性能的角度来说，造成了一定的系统资源浪费。在下一章中，将提出一组新的算法解决上述问题，使算法既在理论上达到Liu&Layland资源利用率界限又具有良好的平均实时性能。

第7章 准划分固定优先级算法的参数化资源利用率界限

上一章提出了一组能够达到 Liu&Layland 资源利用率界限的准划分固定优先级调度算法 RMTS-1 和 RMTS-2。算法中任务划分具有线性复杂度，因此具有非常高的执行效率。但是，RMTS-1 和 RMTS-2 由于使用 Liu&Layland 资源利用率界限作为每个处理器上所能分配任务可调度性测试的唯一依据，因此每个处理器只能使用由 Liu&Layland 资源利用率界限所限制的系统资源，从系统评价性能的角度来说，造成了一定的系统资源浪费。

在系统设计阶段，如果知道更多关于任务集参数的信息，就有可能获得更高的关于已知任务参数的参数化资源利用率界限。参数化资源利用率界限一个著名的例子是谐波任务集 100%的资源利用率界限[4]：如果一个谐波任务集的正规化资源利用率不超过 100%，则其在单处理机上使用 RMS 一定是可调度的。即使整个任务系统不是谐波系统，依然可以通过发掘系统中的“谐波链”来获得更高的资源利用率界限。总的来说，在系统设计阶段，通常可以通过使用已知的系统参数信息来获得更高的资源利用率界限来更好地利用系统资源并降低系统成本。如下文所述，在单处理器调度中存在许多不同形式的参数化资源利用率界限。

这自然就引起了这样一个有趣的问题：是否可以将单处理机调度中各种各样的参数化资源利用率界限也推广到多处理机调度中去？例如，如果给定一个资源利用率总和不超过 M 的谐波任务集，是否可以找到一种调度算法保证此任务集可以在 M 各处理器的平台上可调度？

本章将通过提出一组新的准划分固定优先级调度算法来回答以上问题。即使有了上一章中成功将 Liu&Layland 资源利用率界限推广到多处理机调度工作的启发，将单处理机调度中的参数化资源利用率界限也推广到多处理机调度中仍然是一个难题。这是因为在任务划分过程中任务切分将有可能“创造”新的不符合原任务集参数属性的子任务，因此使针对原任务集参数属性的资源利用率界限不再可用。在第 7.1 节中将详细介绍这个问题。这一章的主要贡献在于通过提出一组新的准划分固定优先级调度算法来解决上述问题，使得单处理机调度中绝大多数都被推广到了多处理机调度中。

本章使用的方法是一种通用的方法：它不受限于某个参数化资源利用率界限的具体形式。该方法所受到的唯一限制是当任务集中单个任务的资源利用率过大时，所适用的

参数化资源利用率界限不能超过一个特定的上限。除此之外，任意根据单处理机 RMS 所得的参数化资源利用界限都可以用来保证多处理机调度的可调度性。具体的说，本章首先提出一个算法 RMTS-RTA-1，其针对每个任务资源利用率不超过 $\Theta(\tau)/(1+\Theta(\tau))$ 的任务集可以将任意单处理机 RMS 的参数化资源利用界限推广到多处理机上。然后提出算法 RMTS-RTA-2，其可以对任意任务集，将单处理机 RMS 的参数化资源利用界限推广到多处理机上，只要该参数化资源利用界限不超过 $2\Theta(\tau)/(1+\Theta(\tau))$。其中 $\Theta(\tau)$ 为 Liu&Layland 资源利用率界限，当其取极限值 $\Theta(\tau)=69.3\%$ 时，$\Theta(\tau)/(1+\Theta(\tau))=40.9\%$，$2\Theta(\tau)/(1+\Theta(\tau))=81.8\%$。

本章提出的算法与上一章的算法相比，除了能达到更高的资源利用率界限，还可以提高系统平均情况下的性能。上一章所提出的算法由于使用 Liu&Layland 资源利用率界限作为每个处理器上所能分配任务可调度性测试的唯一依据，因此每个处理器只能使用由 Liu&Layland 资源利用率界限所限制的系统资源。然而本章提出的算法使用基于响应时间分析（RTA）的精确可调度性测试来判定每个处理器上可以容纳的最大负载。在单处理机上已知 RMS 在平均情况下能够达到 88%的系统资源利用率，这优于其最坏情况的极限值 69.3%。相似地，本章提出算法的平均情况下性能大大优于上一章的算法。这一性能优势是以较高的运行复杂度为代价的：本章提出算法的任务划分算法复杂度是伪多项式的。

7.1　可收缩参数化资源利用率界限

在单处理机上，一个任务集 τ 在调度算法 A 下的一个参数化资源利用率界限（Parametric Utilization Bound，简写为 PUB）是将一个函数 $\Omega(\cdot)$ 应用于 τ 的参数所得到的结果值，使得该值可以保证如果 τ 的资源利用率总和 $U(\tau)$ 满足 $U(\tau)\le\Omega(\tau)$，则在调度算法 A 下，τ 中所有的任务都可以满足其截止期限制。可以通过用使用正规化的资源利用率总和 $U_M(\tau)$ 代替 $U(\tau)$ 来将上述概念扩展到多处理机调度的情况。

在单处理机系统上已知许多单调速率算法下的 PUB。以下为几个典型的例子：

- 著名的 Liu&Layland 资源利用率界限 $\Theta(\tau)$ 本身就是是一个以任务个数为参数的 PUB：$\Theta(\tau)=N\left(2^{1/N}-1\right)$
- 谐波链界限（harmonic chain bound）：$\text{HC-Bound}(\tau)=K\left(2^{1/K}-1\right)$ [130]，其中 K 是任务集中谐波链的数目。著名的谐波任务集界限（100%）是谐波链界限在 $K=1$ 时的特殊情况。

- T 界限，即 T-Bound(τ) [131]是一个关于任务个数与各个任务周期的 PUB：

$$\text{T-Bound}(\tau)=\sum_{i=1}^{N}\frac{T'_{i+1}}{T'_i}+2\cdot\frac{T'_1}{T'_N}-N,$$

其中T'_i 是任务τ_i的缩放后周期[131]。

- R 界限，即 R-Bound(τ)[131]，其与 T-Bound(τ) 相似，不同之处是其使用了更为抽象的参数 r：

$$\text{R-Bound}(\tau)=(N-1)\left(r^{1/(N-1)}-1\right)+2/r-1$$

其中 r 是任务集中所有任务最大与最小缩放后周期的比值。

通过观察上述 PUB 可以发现它们有一个共同的性质：令$\Omega(\tau)$为一个关于任务集τ参数的 PUB，如果通过减小τ中某些任务的最坏情况执行时间参数而获得一个新的任务集τ'，则$\Omega(\tau)$对τ'同样是一个有效的资源利用率界限。满足上述性质的 PUB 称为可收缩（Deflatable）PUB，简写为 D-PUB。下面是 D-PUB 的定义。

定义 7.1（D-PUB）：一个可收缩 PUB，简称 D-PUB$\Omega(\tau)$，是一个满足可收缩性质的 PUB：通过减小任务集τ中某些任务的最坏情况执行时间而获得一个新的任务集τ'，如果τ'满足$U(\tau')\le\Omega(\tau)$，则可以保证τ'在单处理机上可以被 RMS 调度。该性质被称为可收缩性。

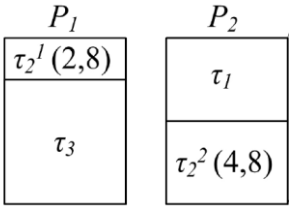
需要注意的是上述定义中的可收缩性与可调度性判定条件的可持续性是不同的。可收缩性不要求原任务集τ满足条件$U(\tau)\le\Omega(\tau)$。事实上，由于τ将要在多个处理器上被调度，一般来讲τ的资源利用率总和是大于 100%，因而大于$\Omega(\tau)$的。$\Omega(\tau)$只是一个通过将τ的参数带入函数$\Omega(\cdot)$而获得的值，而这个值将被用来限制多处理机系统中每个单独的处理器的资源利用率。

可收缩性对于 PUB 是一个非常普遍的性质：事实上，目前已知的所有单处理机 RMS 的 PUB 都满足可收缩性。在下文，将使用$\Omega(\tau)$来表示任意一个在单处理机上 RMS 调度下关于τ参数的 D-PUB。

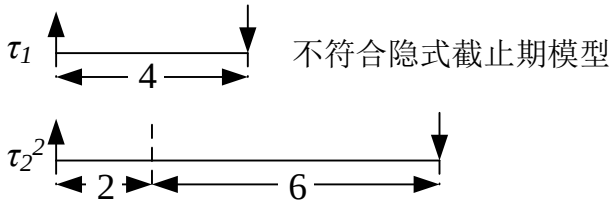
资源利用率界限的可收缩性对于基于划分的多处理机调度具有重要意义，这是因为任务集τ将要被划分成若干个子集，而每个子集将要在每个处理器上单独执行。除此之外，为了超过 50%的资源利用率界限需要进行任务切分，因此一个任务可能会被切分成若干个子任务，其中每个子任务将执行原任务最坏情况执行时间的一部分。可见可收缩性是$\Omega(\tau)$在多处理机上成立的必要条件。

	C_i	T_i
τ_1	2	4
τ_2	6	8
τ_3	12	16

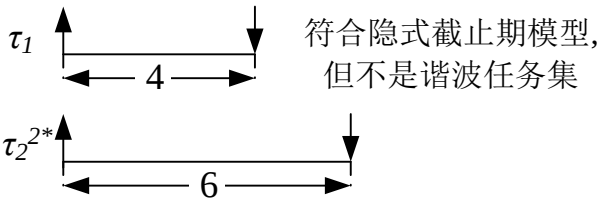
(a) 例子任务集



(b) 划分结果



(c) 处理器 P2 上的任务集不在是隐式截止期



(d) 处理器 P2 上的任务集转化为隐式截止期后部再世谐波任务集

图 7.1 一个谐波任务集在划分后不再是谐波任务集

Fig. 7.1 Partitioning a harmonic task set results in a nonhormonic task set

但是，可收缩性本身并不足以对于将$\Omega(\tau)$推广到多处理机。例如，假设图 7.1(a)中的谐波任务集通过图 7.1(b)的方式进行分配，其中τ_2被切分为τ_2^1和τ_2^2。为了能够正确地执行τ_2，τ_2^1和τ_2^2的执行必须通过同步来保证每次任务释放后τ_2^2在τ_2^1执行结束之后才开始执行。这可以被看作将τ_2^2的相对截止期缩短，如图 7.1(c)所示。这种情况下，τ_2^2不符合$L\&L$任务模型（要求相对截止期与其周期相等），因此任何$L\&L$任务模型的资源利用率界限都不适用于处理器P_2。在[13]中，这个问题的解决方法是使用τ_2^2的相对截止期来代表其周期，如图 7.1(d)所示。这种方法将P_2上的任务集$\{\tau_1,\tau_2^2\}$转化为一个符合$L\&L$任务模型的任务集$\{\tau_1,\tau_2^{2*}\}$。然而这个方法并不能保证对任何 PUB 都普遍适用。例如，上述例子的目标是使针对谐波任务集的 100%资源利用率界限依旧成立，但是如果使用τ_2^2的相对截止期 6 来代表它的周期 8，则$\{\tau_1,\tau_2^{2*}\}$不是一个谐波任务集，因此不能使用 100%的资源利用率界限。本章提出的算法将会解决以上问题。

7.2 针对轻型任务集的算法 RMTS-RTA-1

本节将介绍本章的第一个算法 RMTS-RTA-1。对一个轻型任务集τ，RMTS-RTA-1

可以达到任意的 D-PUB。这里对轻型任务集的定义与上一章相同。首先回顾一下已经在上一章中使用过的相关概念和符合。每个切分任务都被切分一个后部子任务（最后一个子任务）和若干个前部子任务（除最后一个以外的子任务）。用 τ_i^k 来表示一个切分任务 τ_i 的第 k 个子任务，并定义 τ_i^k 的实际相对截止期为

$$\Delta_i^k = T - \sum_{l\in[1,k-1]} R_i^l \tag{7.1}$$

于是，可以用一个三元组 $\langle c_i^k, T_i, \Delta_i^k \rangle$ 来代表一个子任务 τ_i^k，其中 c_i^k 是 τ_i^k 的最坏情况执行时间，T_i 是 τ_i^k 的原始周期，Δ_i^k 是 τ_i^k 的虚拟相对截止期。为了保持一致性，对于每一个非切分任务也可以用一个单独的子任务 τ_i^1 来表示，其中 $c_i^1 = C_i$ 且 $\Delta_i^1 = T_i$。

与上一章中的 RMTS-1 相似，RMTS-RTA-1 可以被概括为两个主要原则：（1）根据任务优先级由低到高的顺序进行任务分配；（2）根据“Worst-Fit-First”原则来选择处理器进行任务分配，以使得各个处理器上的资源利用率平衡地增长。通过这种方式，当任务切分发生时，各个处理器上已经被分配了比较多的低优先级任务，因此被切分任务将在相应的处理器上具有较高的优先级，这有利于每个处理器来容纳更高的资源率利用率。

7.2.1 算法描述

与上一章中的 RMTS -1 相似，RMTS-RTA-1 的划分算法可以被概括为如下规则：

- 根据优先级由低到高的顺序进行任务分配。
- 在算法的每一步，总是选择目前为止所有处理器中已经被分配任务资源利用率总和最小的处理器来接纳当前任务。
- 如果当前任务被分配到一个处理上以后，该处理上包括该任务在内的所有任务依然可以满足其相对截止期，则可以将当前任务整个放到该处理器上。
- 如果当前任务不能够被整个放置到选定的处理器上，则将其切分成两部分，并将其第一部分被分配到当前选定的处理器上，而第二部分留到算法的下一步分配到其它处理器上。任务的切分的原则是，在保证当前选定处理器上没有截止期错失的前提下使第一部分尽量大。

RMTS-RTA-1 与上一章中的 RMTS-1 算法最重要的区别在于，在判断选定的处理器上的任务是否会发生截止期错失时，RMTS-RTA-1 使用精确的响应时间分析，而 SPA1 使用悲观的 L&L 资源利用界限判断。RMTS-RTA-1 使用精确的响应时间分析是其能够达到比 L&L 资源利用界限更高 D-PUB 的关键。另一方面，由于 RMTS-RTA-1 使用精确

的响应时间分析，这使得各个处理器上最后分配所得的资源利用率总和不相同，这对证明算法的资源利用率界限带来了显著的困难。

下面对算法 RMTS-RTA-1 进行详细的描述。算法 1 和 2 是 RMTS-RTA-1 的伪代码。算法一开始，所有任务根据优先级从低到高的顺序进行排列，并将所有的处理器标记为“non-full”，这表示这些处理器还可以接纳更多的任务。在算法的每一步，按顺序选择下一个任务（目前优先级最低的未分配任务），并选择目前为止已分配任务资源利用率总和最小的处理器，然后调用 Assign 例程来进行任务分配。Assign 例程首先检验在加入当前任务以后，选定的处理器上所有的任务是否依旧都能满足截止期。此检验是通过计算每个任务 τ_j^k 的响应时间 R_j^k 来完成的：

$$R_j^k = \sum_{\substack{\tau h \in \tau(P_q) \\ h<j}} \left\lceil \frac{R_j^k}{T_h} \right\rceil C_h + C_j^k$$

由上式得到的每个任务 τ_j^k 的响应时间将和其相对实际截止期 Δ_j^k 相比较。如果该处理器 P_q 上的每个任务的响应时间都不超过其实际截止期，则说明 τ_i^k 可以整个地被分配到 P_q 上。需要注意的是，一个任务的实际相对截止期 Δ_j^k 可能与其周期不同。在介绍完 RMTS-RTA-1 的划分算法整体以后，将讨论如何计算每个任务的实际相对截止期 Δ_j^k。

如果 τ_i^k 不能够被整个分配到选定的处理器 P_q，则使用 $\text{MaxSplit}\left(\tau_i^k, P_q\right)$ 例程将其分为两部分。在不引起 P_q 上任何任务截止期错失的前提下，$\text{MaxSplit}\left(\tau_i^k, P_q\right)$ 将使切分后的第一个子任务尽量大。为了严格地定义 $\text{MaxSplit}\left(\tau_i^k, P_q\right)$ 的这个性质，下面首先介绍瓶颈任务的概念：

1:Task order $\tau_N^1, \cdots, \tau_1^1$ by increasing pririties
2:Mark all processors as *non-full*
3:**while** there is an *non-full* processor **and** an unassigned task **do**
4:　Pick next task τ_i^k
5:　Pick *non-full* processor P_q with minimal $U\left(P_q\right)$
6:　Assign $\left(\tau_i^k, P_q\right)$
7:**end while**
8:If there is an unassigned task, the algrithm fails, otherwise it succeeds.

图 7.2 RMTS-RTA-1 划分算法的伪代码
Fig. 7.2 Pseudo code of the partitioning algorithm of RMTS-RTA-1

1:**if** $\tau\left(P_q\right)$ with τ_i^k is still schedulable **then**
2: Add τ_i^k to $\tau\left(P_q\right)$
3:**else**
4: Split τ_i^k via $\left(\tau_i^k,\tau_i^{k+1}\right):=$ MaxSplit$\left(\tau_i^k,P_q\right)$
5: Add τ_i^k to $\tau\left(P_q\right)$
6: Mark P_q as full
7: τ_i^{k+1} is next task
8:**end if**

图 7.3 Assign()例程

Fig. 7.3 The Assign() routine

定义 7.2：处理器P_q的瓶颈任务是分配到P_q上满足下述性质的的一个任务：如果将P_q上最高优先级任务的最坏情况执行时间增加任意小的正数值，则该任务将不可调度。

一个处理器上可能有多个瓶颈任务。RMTS-RTA-1 根据任务优先级由低至高的顺序进行任务分配，MaxSplit 例程总是操作于某个处理器上最高优先级的任务，因此可以通过如下方式定义 MaxSplit：

定义 7.3：MaxSplit$\left(\tau_i^k,P_q\right)$例程将$\tau_i^k$切分成满足下述条件的两个子任务$\tau_i^k$和$\tau_i^{k+1}$：

将τ_i^k被分配到处理器P_q以后不会使P_q上任何任务在 RMS 不可调度。

P_q接纳τ_i^k以后有至少一个瓶颈任务。

MaxSplit 有多种实现方法。比如，可以对$[0,C_i^k]$进行二分搜索，来找到使P_q上所有任务都满足截止期的τ_i^k执行时间最大值。文献[54]中介绍了一个更高效的 MaxSplit 实现方法，这个方法只需要对$[0,C_i^k]$中的少量值进行检查。虽然文献[54]中方法的计算复杂度依旧伪多项式的，但其实际效率非常高。

RMTS-RTA-1 划分算法中 while 循环的终止条件是所有的处理器都被标志为“full”或者所有的任务都已经被分配完毕了。如果循环是因为前者终止且依旧存在未分配的任务，则说明该任务集不能被成功划分，否则划分成功。

在以上的算法介绍中还有一个未解决的问题，即如何计算每个任务的实际相对截止期Δ_i^k。现在介绍如何解决这个问题。如果τ_i^k是一个非切分任务，则它的实际相对截止期就等于它的周期T_i。下面介绍τ_i^k为切分任务的一个子任务的情况。因为算法是根据任务优先级由低到高的顺序进行任务分配，所以当一个任务被切分且其第一部分（其为一个前部子任务）被分配到一个处理器以后，它的优先级是该处理器上最高的。此后该处理器被标记为“full”而不会再有其它高优先级任务被分配到该处理器上。由此可知：

引理 7.1：一个前部子任务在其宿主处理器上具有最高优先级。

根据此引理可知，每个前部子任务的响应时间等于其执行时间。因此，可以通过将式(7.1)中的R_i^l替换为C_i^l来计算每个子任务的实际相对截止期。其中重要的是计算后部子任务的实际相对截止期（因为一个前部子任务总是具有其宿主处理器上的最高优先级，所以它们肯定总是可调度的，因此不必去担心它们的实际相对截止期是多少）。对于后部子任务实际相对截止期的计算可以被总结为如下引理：

引理 7.2：令τ_i为一个切分任务，其被切分为B_i个前部子任务$\tau_i^{b_1},...,\tau_i^{b_{B_i}}$并分别分配到处理$P_{b_1},...,P_{b_{B_i}}$上，以及一个被分配到处理器$P_t$上的后部子任务$\tau_i^t$。则后部子任务$\tau_i^t$的实际相对截止期$\Delta_i^k$为

$$\Delta_i^t = T_i - \sum_{j\in[1,B_i]} C_i^{b_j} \tag{7.2}$$

以上是 RMTS-RTA-1 的任务划分算法。下面介绍 RMTS-RTA-1 的运行时调度算法。RMTS-RTA-1 在每个处理器上根据所分配任务的原始周期使用 RMS 进行任务调度。此外，每个切分任务的子任务要遵循其前后顺序：即每次任务释放，一个切分子任务只有在它的前继子任务τ_i^{k-1}完成以后才能开始执行。

通过上述对 RMTS-RTA-1 的描述，可以很容易看出 RMTS-RTA-1 划分算法的成果一定能够保证运行时每个任务都能满足其截止期要求。其中重要的是计算后部子任务的实际相对截止期（因为一个前部子任务总是具有其宿主处理器上的最高优先级，所以它们肯定总是可调度的，因此不必去担心它们的实际相对截止期是多少）。对于后部子任务实际相对截止期的计算可以被总结为如下引理：

引理 7.3：任意被 RMTS-RTA-1 的划分算法成功划分的任务集在 RMTS-RTA-1 的调度算法下都是可调度的。

7.2.2 资源利用率界限

本节将证明 RMTS-RTA-1 对于任意轻型任务集都可以达到任意的 PUB$\Omega(\tau)$。首先介绍证明的主要结构。

为了证明RMTS-RTA-1的资源利用率解析$\Omega(\tau)$，首先要证明如果一个轻型任务集τ不可调度，即不能够被 RMTS-RTA-1 的话发算法成功划分，则在划分算法结束后所以处理器上所分配任务的资源利用率总和至少为$M\cdot\Omega(\tau)$。为了证明这个命题，假设某个处理器上分配任务的资源利用率总和严格小于$\Omega(\tau)$，然后证明这将意味着该处理器上不存在瓶颈任务。因为每个进行过 MaxSplit 操作的处理器上存在至少一个瓶颈任务，这导致矛盾。下面，假设P_q为一个所分配任务的资源利用率总和$U(P_q)<\Omega(\tau)$得处理器。P_q

上的任务是非切分任务，前部子任务或后部子任务中的一种。下面对每种情况进行讨论，证明P_q上的任意一种任务都不是瓶颈任务。首先从比较简单的非切分任务和前部子任务开始（引理 7.4），然后再证明后部子任务的情况（引理 7.6）。

引理 7.4：假设任务集τ不能被 RMTS-RTA-1 成功划分，且在划分算法结束后P_q满足$U(P_q)<\Omega(\tau)$，则P_q的瓶颈任务即不是非切分任务也不是前部子任务。

证明：首先根据引理 1 可知P_q上仅有一个前部子任务，且具有P_q上所有任务中的最高优先级。因此，这个前部子任务一定不是瓶颈任务。

下面考虑非切分任务。P_q上的任务是根据其原始周期使用 RMS 进行调度的，且非切分任务的相对截止期等于其原始周期。用Γ表示分配到P_q上的任务集合，然后对应Γ构造这样一个任务集Γ^*：Γ与Γ^*中的任务一一对应；Γ中每个非切分任务在Γ^*中有一个完全相同的对应任务；Γ中每个切分子任务在Γ^*中的对应任务的相对截止期等于其原始周期。因为Γ^*符合隐式截止期任务模型，且可以被看作是通过减小τ中某些任务的执行时间而得到的任务集（有些任务的执行时间被减为 0，这意味着此任务从任务集中被删除）。因此，作为一个 D-PUB，$\Omega(\tau)$可以用来验证Γ^*的可调度性。因为$U(P_q)<\Omega(\tau)$，如果P_q上最高优先级任务的执行时间增加一个任意小的值ε且使P_q上所有任务资源利用率总和依旧不超过$\Omega(\tau)$，Γ^*中的任务依旧可调度。因为Γ与Γ^*的唯一区别在于其中切分子任务的相对截止期可能不同，而由于 RMS 调度是与任务相对截止期无关的，所以可以得出结论：在P_q上最高优先级任务的执行时间增加ε后Γ中的非切分任务依旧是可调度的。注意，在上述证明中只要求保证非切分任务的可调度性，而不必关心那些相对截止期被缩短了的后部子任务是否能够满足截止期。　□

下面将证明对于一个轻型任务集，一个所分配任务资源利用率总和低于$\Omega(\tau)$的处理器上的瓶颈任务也不是后部子任务。证明主要分为两步：首先在引理 5 中将建立保证一个后部子任务不是瓶颈任务的一般性条件，然后应用此结论通过在引理 6 中证明一个资源利用率总和低于$\Omega(\tau)$的处理器上的瓶颈任务不是后部子任务。

先介绍一些将要使用的符号。令τ_i为一个切分任务，其被切分成B_i个前部子任务$\tau_i^{b_1},\tau_i^{b_2},...,\tau_i^{b_{B_i}}$并分别被分配到处理器$P_{b_1},P_{b_2},...,P_{b_{B_i}}$上，和一个被分配到$P_t$上的后部子任务$\tau_i^t$。$\tau_i^t$的资源利用为$U_i^t=C_i^t/T_i$，一个前部子任务$\tau_i^{b_j}$的资源利用率为$U_i^{b_j}=C_i^{b_j}/T_i$。用$U_i^{body}$来表示所有前部子任务资源利用率之和，即

$$U_i^{body}=\sum_{j\in[1,B_i]}U_i^{b_j}=U_i=U_i^t$$

对于每个前部子任务 $\tau_i^{b_j}$，令 X_{b_j} 表示 P_{b_j} 上所分配的优先级低于 $\tau_i^{b_j}$ 的任务资源利用率之和。对于后部子任务 τ_i^t，令 X_t 表示 P_t 上所分配的优先级低于 τ_i^t 的任务资源利用率之和，令 Y_t 表示 P_t 上所分配的优先级高于 τ_i^t 的任务资源利用率之和。

引理 7.5：令 τ_i^t 为分配到 P_t 上的一个后部子任务，$\Theta(\tau)$ 为 Liu&Layland 资源利用率界限。如果满足条件

$$Y_t + U_i^t < \Theta(\tau)\cdot\left(1-U_i^{body}\right) \tag{7.3}$$

则 τ_i^t 一定不是 P_t 上的瓶颈任务。

证明：使 P_t 上最高优先级任务的资源利用率增加 ε，使其满足：

$$\left(Y_t+\varepsilon\right)+U_i^t < \Theta(\tau)\cdot\left(1-U_i^{body}\right)$$

根据 U_i^{body} 和 Δ_i^t 的定义，上式可以改写成：

$$\left(\left(Y_t+\varepsilon\right)+U_i^t\right)\cdot T_i \big/ \Delta_i^t < \Theta(\tau)$$

可以证明，此条件将保证 τ_i^t 依旧可调度（证明过程与上一章引理 6.5 中的证明相同，即通过构造一个新任务集 Γ^* 并证明它的可调度性来间接证明原任务集 Γ 的可调度性）。因此 τ_i^t 不是 P_t 上的瓶颈任务。　□

注意上述引理中使用 Liu&Layland 资源利用率界限 $\Theta(\tau)$，而非更高的 $\Omega(\tau)$ 来保证 τ_i^t 的可调度性。这是因为在 Γ^* 的构造中某些任务的周期被改变了，所以 $\Omega(\tau)$ 可能并不适用于 Γ^*（$\Omega(\tau)$ 的可收缩性只能容忍改变任务的执行时间而非周期）。例如，假设原任务集 Γ 为谐波任务集，Γ^* 则可能不是谐波任务集，这是因为其中某些任务的周期被缩短至 Δ_i^t 而与其它任务的周期不具备谐波关系。但是，无论如何 $\Theta(\tau)$ 对于新任务集 Γ^* 依旧适用，这是因为 $\Theta(\tau)$ 只取决于任务个数 N（关于 N 单调递减）。

接下来，将通过证明任何所分配任务资源利用率之和小于 $\Omega(\tau)$ 的处理器上的后部子任务一定满足条件(7.3)来证明其不可能为瓶颈任务。

引理 7.6：假设任务集 τ 不能被 RMTS-RTA-1 成功划分，且切分任务 τ_i 的后部子任务 τ_i^t 为被分配到处理器 P_t 上，如果 P_t 满足

$$U\left(P_q\right) < \Omega(\tau) \tag{7.4}$$

则 τ_i^t 不是 P_t 的瓶颈任务。

证明：用反证法证明。假设存在一个或多个切分任务不满足此引理，令 τ_i 为所有不满足此引理的任务中优先级最低的任务，即 τ_i^t 为其宿主处理器 P_t 的瓶颈任务，且所有比

τ_i 优先级低的后部子任务要么不是瓶颈任务，要么其宿主处理器上所分配的任务资源利用率总和至少为 $\Omega(\tau)$。

$\tau_i^{b_1},\tau_i^{b_2},...,\tau_i^{b_{B_i}}$ 是 τ_i 的 B_i 个前部子任务，它们的宿主处理器分别为 $P_{b_1},P_{b_2},...,P_{b_{B_i}}$。根据引理 2 可知一个前部子任务一定是其宿主处理器上优先级最高的任务，且任务是根据优先级由低到高的顺序进行分配的，所以处理器 $P_{b_1},P_{b_2},...,P_{b_{B_i}}$ 上的后部子任务的优先级一定低于 τ_i。

接下来首先证明处理器 $P_{b_1},P_{b_2},...,P_{b_{B_i}}$ 各自所分配任务的资源利用率总和至少为 $\Omega(\tau)$。通过反证法来证明这个命题：假设存在一个 P_{b_j} 其上所分配任务的资源利用率总和小于 $\Omega(\tau)$。根据上面的讨论可知，P_{b_j} 上的后部子任务一定不是瓶颈任务。根据引理 7.4 又知道 P_{b_j} 上的非切分任务和前部子任务也一定不是瓶颈任务，这意味着 P_{b_j} 上没有瓶颈任务，而这将导致矛盾，因为每个处理器上至少存在一个瓶颈任务。所以关于 P_{b_j} 上所分配任务的资源利用率之和小于 $\Omega(\tau)$ 的假设一定为假，因此可知所有容纳 τ_i 的前部子任务的处理器上所分配任务的资源利用率一定至少为 $\Omega(\tau)$，由此可知

$$\sum_{j\in[1,B_i]}\underbrace{\left(U_i^{b_j}+X_{b_j}\right)}_{U\left(P_{b_j}\right)}\geq B_i\cdot\Omega(\tau) \tag{7.5}$$

此外，条件(7.4)可以被改写为

$$X_t+Y_t+U_t<\Omega(\tau) \tag{7.6}$$

结合(7.5)与(7.6)可得：

$$X_t+Y_t+U_t<\frac{1}{B_i}\sum_{j\in[1,B_i]}\left(U_i^{b_j}+X_{b_j}\right)$$

因为 RMTS-RTA-1 的划分算法每一步都选择目前为止已分配任务资源利用率总和最小的处理器进行任务分配，所以可知 $\forall j\in[1,B_i]: X_{b_j}\leq X_t$。因此，上述不等式可以被放松为：

$$Y_t+U_i^t<\frac{1}{B_i}\sum_{j\in[1,B_i]}U_i^{b_j}$$

因为 $B_i\geq 1$ 且 $U_i^{body}=\sum_{j\in[1,B_i]}U_i^{b_j}$，由上式可得：

$$Y_t+U_i^t<U_i^{body}$$

为了得到条件(7.3)进而应用引理 5 来证明 τ_i^t 不是瓶颈任务，现在只需要证明上述不等式的右部小于等于条件(7.3)的右部，即证明：

$$U_i^{body} < \Theta(\tau)\cdot\left(1-U_i^{body}\right)$$

容易看出该条件一定成立，这是因为τ_i是一个轻型任务集，所以一定有

$$U_i^{body} \le \frac{\Theta(\tau)}{1+\Theta(\tau)}$$

至此，已经证明满足条件(7.3)，根据引理 7.5 可知τ_i^t不是瓶颈任务，这与证明最开始的假设相矛盾，因此引理一定成立。 □

现在可以将本节已经证明的结论放在一起来证明 RMTS-RTA-1 对于轻型任务集的资源利用率界限。

定理 7.1：对于轻型任务集，$\Omega(\tau)$为 RMTS-RTA-1 的资源利用率界限，即任意满足一下条件的轻型任务集τ

$$U_M(\tau) \le \Omega(\tau)$$

在 RMTS-RTA-1 下都是可调度的。

证明：假设一个满足条件$U_M(\tau) \le \Omega(\tau)$的轻型任务集在 RMTS-RTA-1 下是不可调度的，即在 RMTS-RTA-1 的划分算法结束时依旧存在未分配的任务。由此可知所以处理器上已分配任务的资源利用率总和一定严格小于$M\cdot\Omega(\tau)$，所以一定存在一个处理器其上已分配任务的资源利用率总和严格小于$\Omega(\tau)$。根据引理 7.4 可知，这个处理器上的瓶颈任务即不是非切分任务也不是前部子任务。根据引理 7.6 可知，这个处理器上的瓶颈任务也不是后部子任务。因此可知这个处理器上没有瓶颈任务而导致矛盾。 □

7.3 面向任意任务集的算法 RMTS-RTA-2

本节将介绍 RMTS-RTA-2，以去掉 RMTS-RTA-1 任务集必须是轻型的限制。将要证明，对于任意任务集τ，任意不大于$2\Theta(\tau)/(1+\Theta(\tau))$的 D-PUB $\Omega(\tau)$都可以保证 RMTS-RTA-2 的可调度性，即$\Omega(\tau)$为 RMTS-RTA-2 的资源利用率界限。这意味着，如果可以根据任务集τ的参数得到一个单处理机 RMS 调度的 D-PUB $\Omega'(\tau)$，则在多处理机上 RM-TS 可以达到资源利用率界限$\Omega(\tau)=\min\left(\Omega'(\tau),2\Theta(\tau)/(1+\Theta(\tau))\right)$，其中$\Theta(\tau)$为$L\&L$资源利用率界限，当$\Theta(\tau)=69.3\%$时$2\Theta(\tau)/(1+\Theta(\tau))=81.8\%$。因此可见，虽然$\Omega(\tau)$需要低于一个特点的上限值$2\Theta(\tau)/(1+\Theta(\tau))$，和 Liu&Layland 资源利用率界限$\Theta(\tau)$相比 RM-TS 还是提供了相当大的获得更高资源利用率界限的空间。为了简化叙述，在本章接下来的内容里假设每个任务的资源利用率都不超过$\Omega(\tau)$，因此，重型任务的资源利用率在$\left(\Theta(\tau)/(1+\Theta(\tau)),\Omega(\tau)\right]$的范围内。这个假设并不会妨碍本节的算法在存

在资源利用率超过$\Omega(\tau)$的任务时依旧能够达到资源利用率$\Omega(\tau)$。这是因为，可以将那些资源利用率超过$\Omega(\tau)$的任务各自分配到一个独享的处理器上，如果可以证明其它处理器上任务的资源利用率界限是$\Omega(\tau)$，则整个系统的资源利用率也至少是$\Omega(\tau)$。

7.3.1 算法描述

RMTS-RTA-2 的划分算法如图 7.4 所示。其主要思想与上一章中的 RMTS-2 非常相似。主要区别在于当判断一个任务是满足截止期时 RMTS-2 使用精确的响应时间分析，而 RMTS-2 使用悲观的资源利用率界限判定条件。

算法一开始，首先将所有的处理器标记为普通处理器和待分配处理器。在算法的第一阶段，先按照优先级由高到低的顺序访问所有的任务，并对每一个任务τ_i使用例程DeterminePreAssign(τ_i)来判断其是否应该被预分配（图 7.5 中所示算法）。$\mathbb{P}^{\triangleright}(\tau_i)$记录到目前为止标记为普通的处理器集合。如果$\tau_i$是重型任务，则检查是否满足条件：

$$\sum_{i<j} U_j \le \left(|\mathbb{P}^{\triangleright}(\tau_i)|-1\right)\cdot\Omega(\tau) \tag{7.7}$$

其中$|\mathbb{P}^{\triangleright}(\tau_i)|$表示$\mathbb{P}^{\triangleright}(\tau_i)$中元素的个数，即到目前为止标记为普通处理器的个数。如果满足此条件，则将这个重型任务预分配到当前选定的处理器上（即当前所有普通处理器处理器中具有最小编号的那个），并将此处理器标记为预分配处理器。如果不满足此条件，则不预分配这个重型任务而将其留至算法的下一个阶段处理。预分配条件(7.7)的直观意义如下：如果比一个重型任务τ_i优先级低的任务的资源利用率总和比较小，则应将其进行预分配。否则它的后部子任务可能在其宿主处理器上具有较低的优先级。

算法的第二阶段将余下的任务分配到普通处理器上。余下的任务为所有的轻型任务和那些不满足预分配条件的那些重型任务。这一阶段的分配策略与 RMTS-RTA-1 相同：根据任务优先级由低到高的顺序进行分配，且每一步总是选择所有普通处理器中目前为止所分配任务资源利用率总和最小的那个进行任务分配（调用例程Assign(τ_i^k, P_q)）。

算法的第三阶段将前两阶段余下的任务分配到标记为预分配的处理器上。本阶段与第二阶段有重要的区别：在第二阶段，任务是根据“最坏适用优先”的策略进行分配，即使各个处理器的资源利用率平均地增长，而在第三阶段，任务是根据“最好适用优先”的策略进行分配。更准确地说，在第三阶段，总是选择所有预分配处理器中编号最大的处理器（也就是所有标记为“未完成”的处理器中具有最低优先级预分配任务的那个处理器），并向上尽可能地分配任务，直至其上出现一个瓶颈任务，然后再转向下一个处理进行任务分配。这个分配策略是下一节中使用归纳法来证明算法的资源利用率界限的关键。

1:Mark all processors as *normal* and *non - full*

// Phase 1 : Pre - assignment
2:Sort all tasks in τ in *decreasing* pririty order
3:**for** each task in τ **do**
4:　Pick next task τ_i
5:　**if** DeterminePreAssign(τ_i) **then**
6:　　Pick the *normal* processr with the minimal index P_q
7:　　Add τ_i to $\tau(P_q)$
8:　　Mark P_q as *pre - assigned*
9:　**end if**
10:**end for**

// Phase 2 : Assign remaining tasks to normal processors
11:Sort all unassigned tasks in *increasing* priority order
12:**while** there is a *non - full* normal processor
　　and en unassigned task **do**
13:　Pick next unassigned task τ_i
14:　Pick the *non - full normal* processor with minimal $U(P_q)$
15:　*Assign*(τ_i^k, P_q)
16:**end while**

// Phase3 : Assign remaining tasks to pre - assigned processors
// Remaining tasks are still in increasing priority order
17:**while** there is a *non - full pre - assigned* processor
　　and an unassigned task **do**
18:　Pick next unassigned task τ_i
19:　Pick the *non - full pre - assigned* processor P_q with the largest index
20:　*Assign*(τ_i^k, P_q)
21:**end while**

22:If there is an unassigned task, the algorithm fails, otherwise it succeeds.

图 7.4 RMTS-RTA-2 划分算法的伪代码
Fig. 7.4 Pseudo code of the partitioning algorithm of RMTS-RTA-2

1:$P^{\triangleright}(\tau_i) :=$ the set of *normal* processors at this moment
2:if τ_i is *heavy* then
3: **if** $\sum_{j>i} U_j \le \left(\left|P^{\triangleright}(\tau_i)-1\right|\cdot\Omega(\tau)\right)$ **then**
4: **return true**
5: **end if**
6:**end if**
7:**return false**

图 7.5 DeterminPreAssign()例程
Fig. 7.5 The DeterminPreAssign() routine

在算法的第二和第三阶段，都使用例程 $\text{Assign}(\tau_i^k, P_q)$ 来进行任务的切分和分配。在 $\text{Assign}(\tau_i^k, P_q)$ 中，每个后部子任务的实际截止期是通过假设相应的前部子任务都具有其宿主处理器上最高优先级来进行计算的，而此性质已经被证明在 RMTS-RTA-1 一定被满足（引理 7.1）。因为算法第二阶段的任务分配方式与 RMTS-RTA-1 相同，所以此性质在算法的第二阶段也一定被满足。然而，目前为止还不清楚此性质是否在算法的第三阶段也一定被满足：在算法的第一阶段已经将预分配任务分配到这些与分配处理器上，所以存在预分配任务比相应处理器上前部子任务优先级高的可能性。在后面的引理 7.13 中将要证明，在出分配处理器上的前部子任务一定具有该处理器上的最高优先级，因此例程 $\text{Assign}(\tau_i^k, P_q)$ 同样保证在算法第三阶段切分任务的可调度性。

在上述三个阶段完成以后，如果依旧存在未分配的任务，则划分算法失败，否则为成功。运行时，每个处理器上的任务根据它们的原始周期使用 RMS 进行调度。与 RMTS-RTA-1 相同，一个切分任务的子任务需要遵循相互之间的顺序关系。如果每个预分配处理器上的前部子任务具有最高优先级，则一个成功被 RMTS-RTA-2 划分的任务集一定是可调度的。后面在引理 7.13 中将要证明，在分配处理器上的前部子任务一定具有该处理器上的最高优先级，因此 RMTS-RTA-2 划分算法的成功一定保证任务集在运行时的可调度性。

7.3.2 资源利用率界限

本节将证明 RMTS-RTA-2 的资源利用率界限。与 RMTS-RTA-1 的资源利用率界限证明相比，现在主要的困难时需要处理重型任务集。RMTS-RTA-1 的资源利用率界限证明是通过说明如果划分算法失败，每个单独的处理器上的资源利用率一定不少于 $\Omega(\tau)$。本节的证明将不会对每个单独的处理器进行证明，而是对一组处理器的资源利用率进行证明。

首先介绍一些用到的符号和概念。假设 K 个重型任务在 RMTS-RTA-2 的第一阶段里被预分配。则所有处理器的集合 $\mathfrak{P}$ 可以被分成预分配处理器集 $\mathfrak{P}^{\mathfrak{P}} := \{P_1,...,P_k\}$ 合普通处理器集 $\mathfrak{P}^{\mathfrak{N}} := \{P_{K+1},...,P_M\}$。使用 $\mathfrak{P}_{\geq q} := \{P_q,...,P_M\}$ 来表示标号至少为 q 的处理器集合。

需要证明，如果 τ 不能被成功地被划分，则划分过程结束后所有处理器上已分配任务的资源利用率总和至少为 $M \cdot \Omega(\tau)$。证明通过归纳法完成：

- **基础步骤**：证明所有普通处理器上分配任务的资源利用率总和至少为 $|\mathfrak{P}^{\mathfrak{N}}| \cdot \Omega(\tau)$，其中 $\mathfrak{P}^{\mathfrak{N}}$ 为所有普通处理器的集合。
- **归纳步骤**：对于任意一个预分配处理器 P_m，根据处理器集合 $\{P_{m+1},...,P_M\}$ 的资源利用率来推出处理器集合 $\{P_m,...,P_M\}$ 的资源利用率：

$$\sum_{P_q \in \mathfrak{P}_{\geq m+1}} \mathfrak{U}(P_q) \geq |\mathfrak{P}_{\geq m+1}| \cdot \Omega(\tau)$$

$$\Rightarrow \sum_{P_q \in \mathfrak{P}_{\geq m}} \mathfrak{U}(P_q) \geq |\mathfrak{P}_{\geq m}| \cdot \Omega(\tau)$$

当 $m=1$ 时，即证明了所有处理器上分配任务的资源利用率总和至少为 $M \cdot \Omega(\tau)$。

下面介绍一个新的概念 $\mathfrak{P}^{\mathfrak{R}}(\tau_i^t)$ 以及它的相关性质。

定义 7.3：令 τ_i 为一个重型切分任务，其后部子任务 τ_i^t 被分配到处理器 P_q 上。定义 τ_i^t 的相关处理器集为：

$$\mathfrak{P}^{\mathfrak{R}}(\tau_i^t) := \begin{cases} \mathfrak{P}^{\mathfrak{N}}, & \text{如果} P_q \text{ 是一个普通处理器} \\ \mathfrak{P}_{\geq q}, & \text{如果} P_q \text{ 是一个预分配处理器} \end{cases}$$

直观上，$\mathfrak{P}^{\mathfrak{R}}(\tau_i^t)$ 是那些容纳比 τ_i 优先级低的普通任务的处理器的集合。关于 $\mathfrak{P}^{\mathfrak{R}}(\tau_i^t)$ 上被分配任务的资源利用率有如下性质：

引理 7.7：令 τ_i^t 为一个分配到处理器 P_t 上的重型任务的后部子任务，如果满足条件：

$$\mathfrak{U}(\mathfrak{P}^{\mathfrak{R}}(\tau_i^t)) < |\mathfrak{P}^{\mathfrak{R}}(\tau_i^t)| \cdot \Omega(\tau) \tag{7.8}$$

则下述条件一定成立：

$$Y_t + U_i^t < \Omega(\tau) - U_i^{body}$$

证明：因为 τ_i 是一个被切分的重型任务（即没有被预分配的重型任务），它一定没有满足预分配条件，即它一定满足预分配条件的逆命题：

$$\sum_{j>i} U_j > \left(\left|\mathfrak{P}^{\triangleright}(\tau_i)\right| - 1\right)\cdot\Omega(\tau) \tag{7.9}$$

将所有低优先级任务的资源利用率总和分成如下两部分：

$$\Psi^{\alpha}(\tau_i) := \sum_{\substack{j>i \\ \tau_j \in \tau\left(\mathfrak{P}^{\mathbb{R}}(\tau_i^t)\right)}} U_j$$

$$\Psi^{\beta}(\tau_i) := \sum_{\substack{j>i \\ \tau_j \in \tau\left(\mathfrak{P}\setminus\mathfrak{P}^{\mathbb{R}}(\tau_i^t)\right)}} U_j$$

包含在 $\Psi^{\beta}(\tau_i)$ 中的任务为那些被预分配到 $\mathfrak{P}\setminus\mathfrak{P}^{\mathbb{R}}(\tau_i^t)$ 上的任务。这是因为在 RMTS-RTA-2 划分算法的第二和第三阶段，任务是根据优先级由低到高的顺序进行分配的而且在 $\mathfrak{P}^{\mathbb{R}}(\tau_i^t)$ 被分配满之前没有其它任务被分配到 $\mathfrak{P}\setminus\mathfrak{P}^{\mathbb{R}}(\tau_i^t)$ 中的处理器上。此外，这些预分配任务都在 $\mathfrak{P}^{\triangleright}(\tau_i)$ 中的处理器上，这是因为在 RM-TS 划分算法的第一阶段任务是根据优先级有高到低的顺序进行划分的（$\mathfrak{P}\setminus\mathfrak{P}^{\mathbb{R}}(\tau_i^t)$ 中的所有处理器上都有优先级比 τ_i 高的预分配任务且不会再容纳其它的低优先级任务）。因此，所以被包括在 $\Psi^{\beta}(\tau_i)$ 中的任务都是那些被预分配到 $\mathfrak{P}^{\triangleright}(\tau_i)\setminus\mathfrak{P}^{\mathbb{R}}(\tau_i^t)$ 中处理器上的预分配任务。此外，由于每个预分配处理器上至多有一个预分配任务，且每个任务的资源利用率至多为 $\Omega(\tau)$（根据在本节开始时的假设），可以得到：

$$\Psi^{\beta}(\tau_i) \le \left(\left|\mathfrak{P}^{\triangleright}(\tau_i)\right| - \left|\mathfrak{P}^{\mathbb{R}}(\tau_i^t)\right|\right)\cdot\Omega(\tau) \tag{7.10}$$

通过用 $\Psi^{\alpha}(\tau_i)+\Psi^{\beta}(\tau_i)$ 替换(7.9)中的 $\sum_{j>i} U_j$ 并应用(7.10)，可以得到：

$$\Psi^{\alpha}(\tau_i) > \left(\left|\mathfrak{P}^{\mathbb{R}}(\tau_i^t)\right| - 1\right)\cdot\Omega(\tau) \tag{7.11}$$

自此已经得到了一个 $\Psi^{\alpha}(\tau_i)$ 的下限。下面，将推导一个 $\Psi^{\alpha}(\tau_i)$ 的上限。考虑 $\mathfrak{P}^{\mathbb{R}}(\tau_i^t)$ 中所有处理器上分配任务的资源利用率。当分配 τ_i 时，所有比 τ_i 优先级低的普通任务都已经被分配到 $\mathfrak{P}^{\mathbb{R}}(\tau_i^t)$ 中的处理器上。与上一节中相同，使用 X_{b_j} 表示处理器 P_{b_j} 上所有优先级低于 τ_i 的任务的资源利用率总和，X_t 表示 P_t 上所有优先级低于 τ_i 的任务的资源利用率总和。此外，用 $\mathfrak{P}^{\mathbb{F}}(\tau_i)$ 表示 $\mathfrak{P}^{\mathbb{R}}(\tau_i^t)$ 中那些并不容纳 τ_i 的任何子任务的处理器集合。对于每个 $P_q \in \mathfrak{P}^{\mathbb{F}}(\tau_i)$，用 X_q 表示 P_q 上所有优先级低于 τ_i 的任务的资源利用率总和。因此，可以将 $\Psi^{\alpha}(\tau_i)$ 写成

$$\Psi^{\alpha}(\tau_i) = X_t + \sum_{j\in[1,B_i]} X_{b_j} + \sum_{P_q\in\mathfrak{P}^{\mathbb{F}}(\tau_i)} X_q \tag{7.12}$$

上式中的两个和项有如下性质：

$$\sum_{j\in[1,B_i]} X_{b_j} = \sum_{j\in[1,B_i]} U\left(P_{b_j}\right) - U_i^{body}$$

$$\sum_{P_q\in\mathfrak{P}^{\mathbb{F}}(\tau_i)} X_q \le \sum_{P_q\in\mathfrak{P}^{\mathbb{F}}(\tau_i)} U\left(P_q\right)$$

将这两个性质应用于(7.12)，可以得到$\Psi^{\alpha}\left(\tau_i\right)$的一个上限：

$$X_t + \sum_{j\in[1,B_i]} U\left(P_{b_j}\right) - U_i^{body} + \sum_{P_q\in\mathfrak{P}^{\mathbb{F}}(\tau_i)} U\left(P_q\right) \ge \Psi^{\alpha}\left(\tau_i\right) \tag{7.13}$$

将(7.11)中$\Psi^{\alpha}\left(\tau_i\right)$的下限与(7.13)中$\Psi^{\alpha}\left(\tau_i\right)$的下限相结合可得：

$$X_t + \sum_{j\in[1,B_i]} U\left(P_{b_j}\right) - U_i^{body} + \sum_{P_q\in\mathfrak{P}^{\mathbb{F}}(\tau_i)} U\left(P_q\right) > \left(\left|\mathfrak{P}^{\mathbb{R}}\left(\tau_i^t\right)\right| - 1\right)\cdot\Omega(\tau) \tag{7.14}$$

与此同时(7.8)可以被改写为：

$$U\left(P_t\right) + \sum_{j\in[1,B_i]} U\left(P_{b_j}\right) + \sum_{P_q\in\mathfrak{P}^{\mathbb{F}}(\tau_i)} U\left(P_q\right) < \left|\mathfrak{P}^{\mathbb{R}}\left(\tau_i^t\right)\right|\cdot\Omega(\tau)$$

将上式应用于(7.14)可得：

$$U\left(P_t\right) - X_t < \Omega(\tau) - U_i^{body}$$

又因为$U\left(P_t\right) = X_t + U_i^t + Y_t$，可得$Y_t + U_i^t < \Omega(\tau) - U_i^{body}$。　□

（1）递归的基础步骤

本小节证明递归的的基础步骤。证明的总体策略如下：假设所有普通处理器上分配的任务资源利用率总和小于期望的界限，由此推出$\mathfrak{P}^{\mathbb{N}}$中一定存在的某个处理器其上没有瓶颈任务而导致矛盾。

首先，引理 7.4 对于 RMTS-RTA-2 下的普通处理器依旧成立，即如果一个普通处理器上分配任务的资源利用率总和小于$\Omega(\tau)$，则这个处理器的瓶颈任务既不是非切分任务也不是前部子任务。这是因为 RMTS-RTA-2 在普通处理器上的划分过程与 RMTS-RTA-1 的划分过程完全相同，所以此处可以使用与引理 7.4 完全相同的方法证明。下面，将集中讨论后部子任务的情况。

引理 7.8：假设在 RM-TS 划分算法第二阶段结束之后依然有未分配任务。令τ_i^t为一个被分配到处理器P_t上的后部子任务。如果下述两个条件都成立：

$$\sum_{P_q\in\mathfrak{P}^{\mathbb{N}}} U\left(P_q\right) < \left|\mathfrak{P}^{\mathbb{N}}\right|\cdot\Omega(\tau) \tag{7.15}$$

$$U\left(P_t\right) < \Omega(\tau) \tag{7.16}$$

则τ_i^t不是P_t的瓶颈任务。

注意引理 7.8 与引理 7.6 的区别：引理 7.8 需要条件(7.15)与(7.16)同时成立，而引理 7.6 只需要条件(7.15)成立。这是因为 RMTS-RTA-1 只处理所有任务为轻型任务的情况，而现在 RM-TS 同样需要处理重型任务。

证明：用反证法证明。假设引理对一个或多个任务为假，且令τ_i为所有这些不满足引理的任务中优先级最低的那个。

与 RMTS-RTA-1 中引理 7.6 的证明相似，首先要证明每个容纳τ_i的前部子任务的处理器上所分配任务的资源利用率至少为$\Omega(\tau)$。通过反证法来证明这一命题。假设$U\left(P_{b_j}\right)<\Omega(\tau)$。因为$\tau_i^t$不满足引理，对于$\tau_i^t$条件(7.15)和(7.16)一定都为真，特别地，条件(7.15)一定为真。根据(7.15)和假设$U\left(P_{b_j}\right)<\Omega(\tau)$可知$P_{b_j}$上的后部子任务一定不是瓶颈任务（因为$P_{b_j}$上的后部子任务的优先级低于$\tau_i$，且$\tau_i$是所有不满足引理的任务中优先级最低的那个，所以$P_{b_j}$上的后部子任务一定满足引理）。根据引理 4（上面已经讨论过引理 7.4 对 RM-TS 中的普通处理器依然成立）可知P_{b_j}上的瓶颈任务既不是非切割任务，也不是前部子任务。所以可以得出P_{b_j}上没有瓶颈任务，因而导致矛盾。至此，可以证明每个容纳τ_i前部子任务的处理器上所分配任务的资源利用率总和至少为$\Omega(\tau)$。

接下来将要证明τ_i^t不是一个瓶颈任务。证明的方法是推出条件(7.3)并将其应用于引理 7.5。τ_i要么是轻型任务要么是重型任务。如果τ_i是轻型任务，则证明过程与引理 6 的证明相同，这是因为 RM-TS 划分算法的第二阶段与 RMTS-RTA-1 的划分算法相同。注意，与引理 7.6 相同，证明τ_i是轻型任务的情况只需要条件(7.15)。

下面考虑τ_i是重型任务的情况，这是该引理证明中的难点。分两种情况讨论：

- $U_i^{body} \geq \dfrac{\Omega(\tau)-\Theta(\tau)}{1-\Theta(\tau)}$

根据定义，因为τ_i^t被分配到普通处理器上，所以$\mathfrak{P}^{\mathbb{R}}(\tau_i)=\mathfrak{P}^{\mathbb{N}}$。又因为$\tau_i$是重型任务，根据条件(7.15)和引理 7.7 可以得到：

$$Y_t+U_i^t<\Omega(\tau)-U_i^{body} \tag{7.17}$$

为了得到引理 7.5 的条件(7.3)来证明τ_i^t不是瓶颈任务，只需要证明

$$\Omega(\tau)-U_i^{body} \leq \Theta(\tau)\left(1-U_i^{body}\right)$$

$$\Leftrightarrow \left(1-\Theta(\tau)\right)U_i^{body} \geq \Omega(\tau)-\Theta(\tau)$$

$$\Leftrightarrow U_i^{body} \geq \frac{\Omega(\tau)-\Theta(\tau)}{1-\Theta(\tau)}\left(\text{since } \Theta(\tau)<1\right)$$

根据此分类的条件，最后一个不等式显然成立。因此对于此分类条件(7.3)一定成立。

- $U_i^{body} < \dfrac{\Omega(\tau)-\Theta(\tau)}{1-\Theta(\tau)}$

首先，条件(7.16)可以被重写成

$$X_t + Y_t + U_i^t < \Omega(\tau) \tag{7.18}$$

上面已经证明，每个容纳τ_i的前部子任务的处理器上所分配任务的资源利用率总和都至少为$\Omega(\tau)$，因此可知：

$$\sum_{j\in[1,B_i]} X_{b_j} + U_i^{body} > B_i \cdot \Omega(\tau)$$

因为在 RMTS-RTA-2 划分算法第二阶段的每一步总是选择目前为止已经分配任务的资源利用率总和最小的那个处理器进行任务分配，所以可知对于每个X_{b_j}都有$X_t \geq X_{b_j}$，所以上式可以转化为

$$B_i X_t + U_i^{body} \geq B_i \cdot \Omega(\tau)$$
$$\Rightarrow X_t \geq \Omega(\tau) - U_i^{body} \left(\text{since } B_i \geq 1\right)$$

将上式与(7.18)相结合可以得到：

$$Y_t + U_i^t < U_i^{body}$$

现在，为了证明条件(7.3)只需要证明

$$U_i^{body} \leq \Theta(\tau)\left(1-U_i^{body}\right)$$

$$\Leftrightarrow U_i^{body} \leq \frac{\Theta(\tau)}{1+\Theta(\tau)}$$

根据此分类的前提条件$U_i^{body} < \dfrac{\Omega(\tau)-\Theta(\tau)}{1-\Theta(\tau)}$，只需要证明

$$\frac{\Omega(\tau)-\Theta(\tau)}{1-\Theta(\tau)} \leq \frac{\Theta(\tau)}{1+\Theta(\tau)}$$

$$\Leftrightarrow \Omega(\tau)-\Theta(\tau)+\Theta(\tau)\cdot\Omega(\tau)-\Theta(\tau)^2 \leq \Theta(\tau)-\Theta(\tau)^2$$

$$\Leftrightarrow \Omega(\tau) \leq \frac{2\Theta(\tau)}{1+\Theta(\tau)}$$

因为在 RMTS-RTA-2 中规定$\Omega(\tau)$不超过$\dfrac{2\Theta(\tau)}{1+\Theta(\tau)}$，所以上面最后一个不等式一定成立。

综上，两种情况下条件(7.3)都成立，因此根据引理 7.5 可知τ_i^t不是瓶颈任务。　□

引理 7.9：假设在 RM-TS 算法第二阶段结束后仍然存在未分配任务，则一定有

$$\sum_{P_q \in \mathrm{P}^{\mathrm{N}}} U\left(P_q\right) \geq \left|\mathrm{P}^{\mathrm{N}}\right| \cdot \Omega(\tau)$$

证明：通过反证法证明。假设

$$\sum_{P_q \in \mathrm{P}^{\mathrm{N}}} U\left(P_q\right) < \left|\mathrm{P}^{\mathrm{N}}\right| \cdot \Omega(\tau)$$

则 P^{N} 中至少存在一个处理器 P_q 满足 $U\left(P_q\right) < \Omega(\tau)$。根据引理 7.5 可知 P_q 上的瓶颈任务既不是非切分任务也不是前部子任务，且根据引理 7.8 可知 P_q 上的瓶颈任务也不是后部子任务。因此可知 P_q 上不存在瓶颈任务，因而导致矛盾，由此引理可证。　□

（2）归纳步骤

首先介绍一个关于预分配任务优先级的性质：

引理 7.10：假设 P_m 是一个预分配处理器，且满足条件

$$\sum_{P_q \in \mathrm{P}_{\geq m+1}} U\left(P_q\right) \geq \left|\mathrm{P}_{\geq m+1}\right| \cdot \Omega(\tau) \tag{7.19}$$

则 P_m 上的预分配任务在 P_m 上所有任务中是优先级最低的。

证明：令 τ_i 为 P_m 上的一个预分配任务。可知 τ_i 一定满足预分配条件：

$$\sum_{j>i} U_j \leq \left(\left|\mathrm{P}^{\triangleright}\left(\tau_i^t\right)\right| - 1\right) \cdot \Omega(\tau)$$

将上式应用于(7.19)可以得到：

$$\sum_{P_q \in \mathrm{P}_{\geq m+1}} U\left(P_q\right) \geq \sum_{j>i} U_j \tag{7.20}$$

这意味着具有较大编号的处理器可以容纳所有低优先级任务。根据 RMTS-RTA-2 的划分算法可知，在具有较大编号的处理器被分满之前，除了预分配任务 τ_i，没有任务被分配到 P_m 上。因此可知，没有比 τ_i 优先级低的任务被分配到 P_m 上。　□

引理 7.11：使用 RMTS-RTA-2 对任务集 τ 进行划分，假设在 P_m 分满之后依旧有未分配任务（P_m 上至少存在一个瓶颈任务）。如果满足条件：

$$\sum_{P_q \in \mathrm{P}_{\geq m+1}} U\left(P_q\right) \geq \left|\mathrm{P}_{\geq m+1}\right| \cdot \Omega(\tau) \tag{7.21}$$

则一定满足

$$\sum_{P_q \in \mathrm{P}_{\geq m}} U\left(P_q\right) \geq \left|\mathrm{P}_{\geq m}\right| \cdot \Omega(\tau)$$

证明：用反证法证明。假设

$$\sum_{P_q \in \mathfrak{P}_{\geq m}} U\left(P_q\right) < |\mathfrak{P}_{\geq m}| \cdot \Omega(\tau) \tag{7.22}$$

根据(7.22)和(7.23)可知：

$$U\left(P_m\right) < \Omega(\tau) \tag{7.23}$$

下面将根据(7.23)来证明 P_m 上的瓶颈不是非切分任务，不是前部子任务，也不是后部子任务，这将导致矛盾而完成证明。下面就对以上三个不同的任务类型进行分别讨论。

首先考虑非切分任务。与上一节的证明类似，资源利用率界限 $\Omega(\tau)$ 的可收缩性保证了非切分任务的可调度性（尽管切分任务的相对截止期会变短）。因此条件(7.23)保证了非切分任务一定不是 P_m 的瓶颈任务。

下面考虑前部子任务。根据引理 7.10 可知 P_m 上的预划分任务具有最低优先级。此外，还知道 P_m 上所有普通任务的优先级都低于前部子任务（因为在 RMTS-RTA-2 的第三阶段任务是根据优先级由低到高的顺序进行分配的）。因此，可以得出 P_m 上的前部子任务具有最高优先级，所以其不可能是瓶颈任务。

最后考虑后部子任务的情况。令 τ_i^t 为一个分配到 P_m 上的后部子任务。进行分类讨论：

$$U_i^{body} < \frac{\Theta(\tau)}{1+\Theta(\tau)}$$

根据归纳假设条件(7.21)和引理 7.10 可知 P_m 上的预分配任务具有最低优先级，因此 X_t 中至少包括这个预分配任务的资源利用率。由于预分配任务一定是重型任务，可知：

$$X_t \geq \frac{\Theta(\tau)}{1+\Theta(\tau)} \tag{7.24}$$

可以将(7.23)重写为 $X_t + Y_t + U_i^t < \Omega(\tau)$，并将其应用于(7.24)获得：

$$Y_t + U_i^t < \Omega(\tau) - \frac{\Theta(\tau)}{1+\Theta(\tau)} \tag{7.25}$$

在 RMTS-RTA-2 中假设 $\Omega(\tau)$ 不超过以下上限：

$$\Omega(\tau) \leq \frac{2\Theta(\tau)}{1+\Theta(\tau)}$$

$$\Leftrightarrow \Omega(\tau) - \frac{\Theta(\tau)}{1+\Theta(\tau)} \leq \Theta(\tau)\left(1 - \frac{\Theta(\tau)}{1+\Theta(\tau)}\right)$$

又因为 $U_i^{body} < \frac{\Theta(\tau)}{1+\Theta(\tau)}$ 可得：

$$\Omega(\tau) - \frac{\Theta(\tau)}{1+\Theta(\tau)} \leq \Theta(\tau)\left(1 - U_i^{body}\right)$$

又根据(7.25)可得$Y_t+U_i^t<\Omega(\tau)\left(1-U_i^{body}\right)$。根据引理 5 可知$\tau_i^t$不是瓶颈任务。

$$U_i^{body}\geq\frac{\Theta(\tau)}{1+\Theta(\tau)}$$

因为$U_i>U_i^{body}\geq\frac{\Theta(\tau)}{1+\Theta(\tau)}$，可知$\tau_i$为重型任务。因为$P_m$是一个预分配处理器，所以$\mathfrak{p}^{\Re}(\tau_i)=\sum_{P_q\in\mathfrak{P}_{\geq m+1}}U(P_q)$，又根据归纳假设(7.21)引理 7.7 可知：

$$Y_t+U_i^t<\Omega(\tau)-U_i^{body}$$

又因为$\Omega(\tau)\leq\frac{2\Theta(\tau)}{1+\Theta(\tau)}$可得

$$Y_t+U_i^t<\frac{2\Theta(\tau)}{1+\Theta(\tau)}-U_i^{body}\tag{7.26}$$

此分类的前提条件$U_i^{body}\geq\frac{\Theta(\tau)}{1+\Theta(\tau)}$可以转换为

$$\frac{2\Theta(\tau)}{1+\Theta(\tau)}-U_i^{body}\leq\Theta(\tau)\left(1-U_i^{body}\right)$$

将其应用于(7.26)可得$Y_t+U_i^t<\Theta(\tau)\left(1-U_i^{body}\right)$。所以根据引理 7.5 可知$\tau_i^t$不是$P_m$上的瓶颈任务。

综上所述，在两种情况下τ_i^t都不是P_m上的瓶颈任务。因此可以推出P_m上不存在瓶颈任务，导致矛盾。　□

（3）资源利用率界限

引理 7.12：任意一个满足$U_M(\tau)\leq\Omega(\tau)$的任务集$\tau$（其中$\Omega(\tau)\leq\frac{2\Theta(\tau)}{1+\Theta(\tau)}$）都能够被 RMTS-RTA-2 的划分算法成功地划分。

证明：根据引理 7.9（归纳法中的基本情况）和引理 7.11（归纳法中的归纳步骤），可以使用归纳法证明如果 RMTS-RTA-2 的划分算法失败，则所有处理器上所分配任务的资源利用率综合至少为$M\cdot\Omega(\tau)$。因为划分算法失败后，依然有还没有分配到任何处理器上的任务，所以τ的资源利用率综合一定严格大于$M\cdot\Omega(\tau)$，即其正规化资源利用率$U_M(\tau)$一定严格大于$\Omega(\tau)$。　□

引理 7.13：任何被 RM-TS 的划分算法的任务集在运行时都可以被 RMTS-RTA-2 的

调度算法成功调度。

证明：与 RMTS-RTA-1 算法相似，RMTS-RTA-2 调度算法下所有普通处理器上任务的可调度性由例程 $\text{Assign}\left(\tau_i^k, P_q\right)$ 保证。在预分配处理器上，$\text{Assign}\left(\tau_i^k, P_q\right)$ 保证任务可调度性的前提条件是每个预分配处理器上的前部子任务具有该处理器上的最高优先级。下面将证明该前提条件为真。

令 P_q 为一个参与 RMTS-RTA-2 划分算法第三阶段的预分配处理器，且 $\tau_i^{b_j}$ 为分配到 P_q 上的前部子任务。根据引理 7.9 和引理 7.11 可以通过归纳法证明 $\mathfrak{P}_{\geq q+1}$ 中的处理器上所分配任务的资源利用率之和至少为 $|\mathfrak{P}_{\geq q+1}| \cdot \Omega(\tau)$，又根据引理 7.10 可知每个预分配任务具有相应处理器上的最低优先级，所有其优先级低于 $\tau_i^{b_j}$。因为任务是根据优先级从低到高的顺序分配且 $\tau_i^{b_j}$ 是最后一个被分配到 P_q 上的任务。综上所述，可知 $\tau_i^{b_j}$ 具有 P_q 上最高优先级。 □

至此，已经证明了任意资源利用率总和低于 $\Omega(\tau)$ 的任务都可以被 RMTS-RTA-2 成功划分且在运行时可调度，因此可以得到 RMTS-RTA-2 的资源利用率界限：

定理 7.2：给定一个根据任务集 τ 的参数得到的单处理机 RMS 调度下的可收缩参数资源利用率界限 $\Omega(\tau) \leq \dfrac{2\Theta(\tau)}{1+\Theta(\tau)}$，如果

$$\mathfrak{U}_M(\tau) \leq \Omega(\tau)$$

则 τ 可被 RMTS-RTA-2 调度。

7.4 小结

本章提出了一组新的准划分固定优先级调度算法，将单处理机固定优先级调度中的绝大部分参数化资源利用率界限都可以推广到多处理机系统。本章首先提出一个算法 RMTS-RTA-1，其针对每个任务资源利用率不超过 $\Theta(\tau)/(1+\Theta(\tau))$ 的任务集可以将任意单处理机 RMS 的参数化资源利用界限推广到多处理机上。然后提出算法 RMTS-RTA-2，其可以对任意任务集推广将单处理机 RMS 的参数化资源利用界限推广到多处理机上，只要该参数化资源利用界限部超过 $2\Theta(\tau)/(1+\Theta(\tau))$。此外，上一章所提出的算法由于使用 Liu&Layland 资源利用率界限作为每个处理器上所能分配任务可调度性测试的唯一依据，而本章的算法然而本章提出的算法使用基于响应时间分析（RTA）的精确可调度性测试来判定每个处理器上可以容纳的最大负载，因此可以大幅度地提高系统平均情况下的性能。

第8章 一种共享缓存敏感的调度算法及分析

在实时系统中使用多核处理器的一个主要的障碍是难以对其上所运行程序的时间属性进行分析与保证。多核处理器上的片上共享资源（如共享缓存）对其上所运行程序的时间属性有重要影响。传统面向单核处理器的实时系统设计与分析方法已经无法解决多核系统中片上共享资源对系统时间可预测性所带来的难题。

本章提出一种新的系统设计方法来解决上述问题。该方法通过使用缓存空间隔离技术来避免多核处理器上各个核上并行运行的实时任务之间之间由共享缓存所引起相互干扰。本章研究了一种基于上述方法的调度算法。该算法在调度任务实时，不但为任务分配处理器资源，同时还根据其需要分配一定数量的共享缓存块，并保证同一时刻并行运行的实时任务之间所分配的缓存块没有重叠，以达到缓存空间隔离并最终避免程序间由共享缓存所引起相互干扰的目的。从技术贡献的角度，本章研究了上述算法的可调度性判定问题。本章首先提出了一个基于线性规划问题求解的可调度性判定方法，然后又对该分析方法进行近似从而得到一个基于封闭表达式的判定方法来提高分析效率。通过使用随机生成的实验表明，本章所提出的基于线性规划问题求解的可调度性判定方法可以在几分钟内分析规模达几千个任务的系统。实验还表明，本章提出的基于封闭表达式的可调度性判定方法在具有极高分析效率的同时，可以获得与第一个分析方法非常相近的分析质量，因此该方法非常适合作为实时系统中运行时准入控制以及迭代的系统设计空间拓展过程。

8.1　片上共享资源与实时调度

多核处理器上的片上共享资源（如共享缓存）对其上所运行程序的时间属性有重要影响。单处理器上的嵌入式实时系统的设计与分析技术已经非常成熟。运用这些技术，可以对每个实时任务的最坏情况执行时间（WCET）进行估计，然后使用单个任务的 WCET 进行系统级的可调度性分析。在任务的最坏情况执行时间分析中一个重要的问题是对程序缓存访问行为进行建模和分析。因为程序执行每条指令的时间依其缓存访问是命中（Hit）还是错失（Miss）有很大不同，所以程序的缓存访问行为对程序的执行时间有巨大影响。在过去 20 多年里，单处理器上缓存的建模与分析问题技术已经发展得非常成熟。目前绝大多数程序最坏情况执行时间分析工具里都支持对缓存的分析功能。

但处理器上现有的技术并不适用于带有共享缓存的多核处理器，原因是在一个处理器核心上运行的程序可能会将在另一个处理器核心上运行的程序所使用的共享缓存内

容置换出去。在这种情况下，就不能对单个程序的最坏情况执行时间进行独立的分析。现有工作中对带有共享缓存的多核处理器进行程序最坏情况执行时间分析的工作都只适用于特定的软件结构和比较简单的硬件体系结构[132~133]。程序最坏情况执行时间分析的研究者们普遍认为对带有共享缓存的多核处理器进行高效和精确的程序最坏情况执行时间分析是极其困难的。

本章所提出的方法不是直接去解决上述难题，而是通过在实时调度中使用缓存划分技术（页着色技术[134]）来隔离并行执行的实时任务的缓存空间，最终避免它们之间的相互干扰。对于多核处理器上其它的共享资源，比如共享总线，可以使用如时间分片，轮转访问，基于优先级的访问等方法来对各个核之间的竞争进行控制[135~136]。文章将只考虑共享缓存造成的核间相互干扰。本章假设整个共享缓存被分为若干个大小相等的缓存块，且假设已知每个程序在运行时所占用的缓存块的个数，且每个任务可以根据其所占缓存块的大小来对其最坏情况执行时间进行分析。在系统设计过程中，设计者可以在使用本章所提出方法的基础上调整任务所使用缓存块的个数及其最坏情况执行时间。本章将使用全局不可抢占固定优先级调度算法作为基础调度算法，但是本章所采用的技术也可以扩展到其它多处理机实时调度算法。

8.2　相关工作

因为一个程序对共享缓存的访问行为对系统性能造成的影响远大于诸如本地缓存和处理器流水线等其它因素[137]，多核处理器上由于共享缓存所造成冲突会极大地降低系统的性能和可预测性。Chandra 等人通过实验表明一个程序的执行时间根据其同时执行的程序的不同（有较高的缓存错失率或较低的缓存错失率）而造成的变化可以达 65%[138]。一个错失率较高的同时执行程序可以让被测程序的执行时间大幅度下降。这种情况可以通过人为地避免那些对存储器访问量较大的程序并行执行而得到改善[137]。Anderson 等人通过在实时调度中鼓励或者避免一些任务的并行执行来在满足系统实时约束的同时增强任务缓存访问的性能[62~63, 139~140]。这些工作主要面向软实时系统，并且假设任务的最坏情况执行时间是事先知道的。但是，尽管上述方法中通过增强任务缓存访问的性能可以降低任务的运行时间，每个任务的最坏情况执行时间的计算方法确是未知的。本章提出的方法也是将对缓存的考虑引入实时调度中来。但是与 Anderson 等人的工作不同，本章通过使用缓存空间隔离来彻底避免同时运行任务间有共享缓存访问而造成的相互干扰，因此可以使用单核处理器系统的任务最坏情况执行时间分析技术来准确高效地对每个程序的执行时间进行估计，并进而应用这些信息进行系统级的可调度性分析。

Yan 和 Zhang 研究了具有共享缓存的多核处理器上的程序最坏情况执行时间分析问题[132]。他们的工作假设一种简单的运行情况：两个程序同时分别运行在一个双核处理器的两个核上，该处理器具有一个共享的直接映射共享缓存。他们的分析方法的存在很大的局限性：首先，现代多核处理器一般使用组相联共享缓存而非直接映射共享缓存；其次，他们的分析方法非常悲观，尤其是在芯片上有更多的处理器核心时；最后，他们的方法不能够处理存在多个实时任务并对它们进行调度的情况。Li 等人将 Yan 和 Zhang 的方法进行了改进，用于处理组相联共享缓存[133]，但是这些工作仍将受到分析精确性和适用任务模型的范围的严重限制。

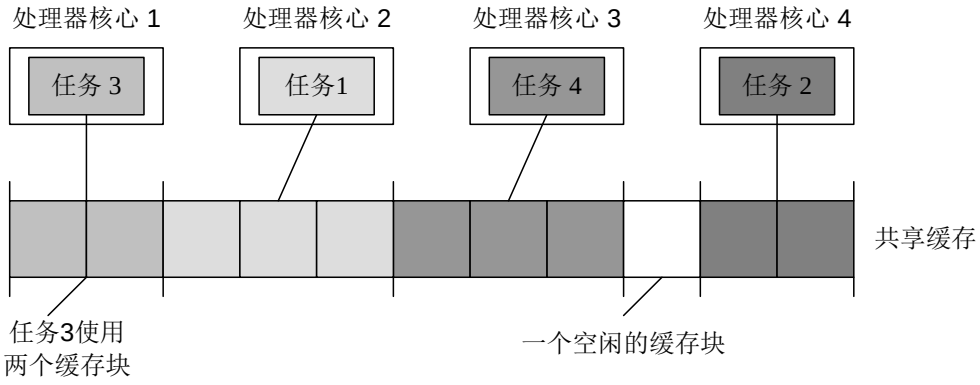
多处理机上全局调度算法以及分析技术被推广到了处理多个运算资源的一般情况，比如在一维动态可重配置 FPGA 上进行任务的调度（每个硬件任务在执行时可以占用多个 FPGA 列）[141~142]。但是，这些调度分析技术并不适用于本章所提出调度算法的分析，因为在本章提出的缓存敏感调度中，任务要在两种不同的资源上同时进行调度（处理器核心和缓存块）。此外，Fisher 等人研究了静态地将周期性任务集在多处理机平台上进行划分的问题[143]，其划分目标是使分配到每个处理器上任务的资源利用率总和都不超过 1，且存储需求量不超过单个处理器上存储器的容量。Suhendra 等人[144]和 Salamy 等人[145]研究了如果将一个任务图划分到片上多处理机。上述工作中研究的都是对任务进行静态的划分问题，而其运行时的可调度性分析问题就退化成了单处理机调度的可调度性分析问题。Suhendra 等人研究了几种基于共享缓存划分和锁定的系统设计方法[146]，但是这些工作没有研究如何进行系统级的可调度性分析。

8.3　片上缓存空间划分

假设一个多核处理器的多个核共享片上缓存（通常是二级或三级片上缓存）。本章将不考虑各个处理器核心的本地缓存（通常是一级片上缓存）及其他共享资源（如片上共享总线等）。因为各个核心对共享片上缓存的并行访问会引起相互干扰而降低程序的可预测性，本章将采用片上缓存划分机制，将片上缓存空间划分成若干个互不重叠的区域来给每个正在运行的任务单独使用（如图 8.1(a)所示）。

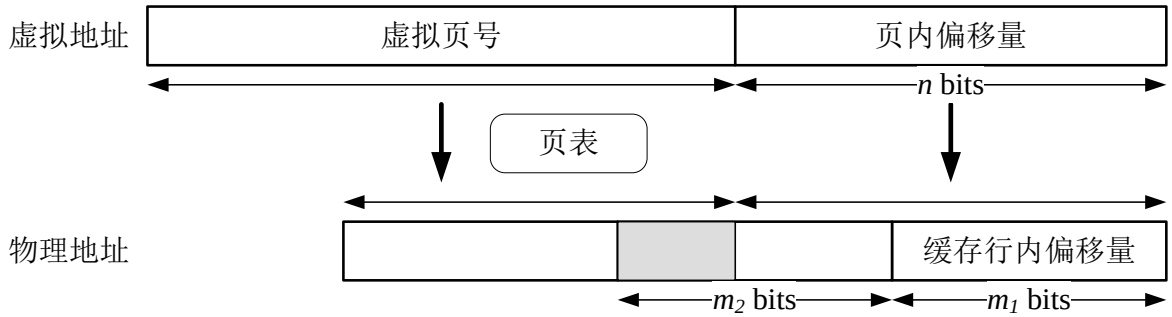
将共享缓存在同时运行的任务之间进行划分曾被用来提高多核系统的平均性能或者在单核处理器上降低任务上下文切换开销来增强系统的可预测性[147~149]。共享缓存划分可以通过不同的方式进行。假设一个 k 路组相联的共享缓存含有 l 个缓存组。共享缓存划分方法可以分为基于组的划分[134]和基于路的划分[147]。基于组的划分方法也称为基于行的划分，其将不同的缓存组划分成不同的缓存块，因此其最多可以划分成 l 个缓存块。基于路的划分也称为基于列的划分，其最多可以划分成 k 个缓存块。此外还可以进

行基于组和基于路相混合的划分。划分方法还可以根据是否需要额外的硬件支持而分为基于软件的划分和基于硬件的划分。



(a) 缓存空间隔离

(a) Cache space isolation



(b) 页着色中的地址映射

(b) Address mapping in page coloring

图 8.1 缓存空间隔离与页着色示意图

Fig. 8.1 Illustration of cache space isolation and page coloring

这里将介绍一种称为“页着色”的基于组的划分方法。这种方法的优点是它完全基于软件方法而不需要任何特殊的硬件支持。该方法通过管理虚拟内存地址与物理内存地址之间的映射来达到缓存划分的目的。假设一个由 2^{m_2} 个缓存组组成的共享缓存，其中每个缓存行由 2^{m_1} 个字组成。在此缓存中，物理地址的最低 m_1 位表示缓存行内的偏移量，且有 m_2 位标识缓存组编号（如图 8.1(b)所示）。此外，假设一个虚拟页面的大小为 2^n 个字，所有虚拟地址中的最低 n 位标识页内偏移量，而其它的字表示页面编号，且将根据页表转换为物理地址。如果 $m_1 + m_2 > n$，则用来寻址缓存组的某些位实际可以通过页分配进行控制。根据此方法，页面颜色的总数为 $2^{(m_1+m_2)-n}$。

通过上述方法进行缓存划分的一个例子是[150]，其通过修改 Linux 内核的页分配算法在一个 Power 5 双核处理器上来实现总共 16 个颜色的页着色缓存空间划分。需要注意的是，这种方法需要在物理内存上强制使用某种固定的布局，因此其对每个任务可以使用的内存空间大小和重新着色的灵活性都有一定限制。这些问题可以通过[151]中提出

的方法来解决。该方法可以通过改变系统主板上处理器与主存之间的连接逻辑来高效地实现重新着色。因此，本章将假设一个共享缓存具有若干个大小相等的缓存块，且这些缓存块可以在运行时任意地分配给某个任务。

可以看出，本章研究的任务模型中，周期性任务 τ_i 除了原有的最坏情况执行时间 C_i，相对截止期 D_i 和周期 T_i 以外，还具有一个新增参数 A_i，即该任务所需要缓存块的数目。

8.4　缓存敏感调度算法

本节将介绍提出的缓存敏感调度算法及相关分析技术背景，并分析为何现有技术不能解决本章提出的缓存敏感调度算法的可调度性分析问题。

8.4.1 资源利用率界限

因为在可抢占调度中由缓存置换造成的上下文切换开销通常难以预测，本章考虑使用非抢占的调度算法。缓存空间切分的思想可以被应用于多种传统多处理机调度算法，本章将使用非抢占固定优先级调度算法作为基础调度算法。

本章提出的缓存敏感非抢占固定优先级调度算法在有足够缓存空间的前提下，总是选择所有就绪任务中最高优先级。具体地说，一个任务实例 J_i 在满足如下条件时将被调度执行：

1. J_i 是目前所有就绪任务中优先级最高的
2. 目前至少有一个处理器空闲
3. 目前至少有 A_i 个缓存块空闲

因为本章假设每个任务都满足 $D_i \le T_i$，因此在任何时候每个任务至多有一个已经释放且未完成的任务实例。

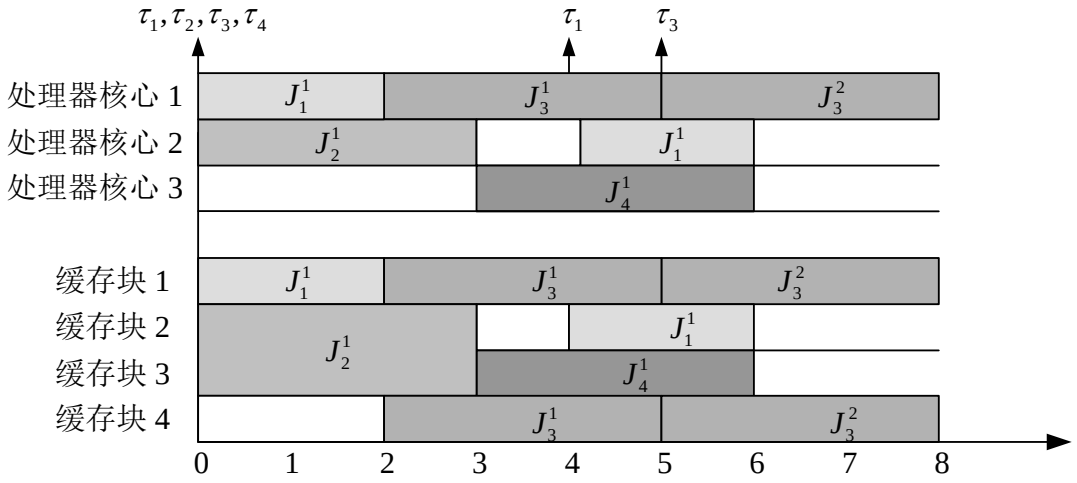
图 8.2(a)表 1 中的任务集被 $\mathrm{FP}_{\mathrm{CA}}$ 调度，其结果如图 8.2(b)所示（其中假设所有的任务同时释放第一个实例）。根据 $\mathrm{FP}_{\mathrm{CA}}$ 的调度规则，在时刻 0 尽管任务实例 J_4^1 有足够的空闲缓存空间和空闲处理器，其仍旧不能开始执行，这是因为它并不是所有就绪任务实例中优先级最高的（存在一个更高优先级的就绪任务实例 J_3^1 在等待执行）。J_3^1 不能开始执行的原因是没有足够的缓存空间。因此，可以看出 $\mathrm{FP}_{\mathrm{CA}}$ 可能会造成资源浪费：如果一个就绪的高优先级的任务实例不能够获得足够的资源开始运行，则系统中所有就绪的低优先级任务无法开始执行，即使现有的空闲资源足够执行这些低优先级任务实例。$\mathrm{FP}_{\mathrm{CA}}$ 虽然可能造成平均情况下的资源浪费，但它维护了比较严格的优先级顺序，因此这有利于提高系统的可预测性。这种调度算法称为阻塞性调度算法。某些情况下，系统设计者可能更倾向于让上述情况中被阻塞的低优先级任务提前执行，以优化系统的性能。这种调

度算法称为非阻塞性调度算法。FP_{CA} 属于阻塞性调度算法，但是本章提出的可调度性分析技术同样也适用于非阻塞性调度算法。在后面的第 X 节中，将详细地对阻塞与非阻塞性调度算法进行比较。

任务	D_i	T_i	C_i	A_i
τ_1	3	3	2	1
τ_2	4	4	3	2
τ_3	5	5	2	2
τ_4	8	8	2	1

(a) 一个例子任务集

(a) A task set example



(b) FP_{CA} 下的调度

(b) Scheduling under FP_{CA}

图 8.2　FP_{CA} 调度示意图

Fig. 8.2 Illustration of FP_{CA}

8.4.2 问题窗口分析框架

为了分析一个任务集能否被 FP_{CA}，将假设一个任务会错失截止期，并分析一个特定的时间区域内其它任务的干涉是否足以使其错失截止期。此分析框架被称为问题窗口分析框架[4]。

下面将讨论使用传统的问题窗口分析框架对 FP_{CA} 的可调度性进行分析所存在的问题。首先将介绍，如果任务只使用处理器核心上或只使用缓存资源的两种特殊情况下使用问题窗口分析框架得到的可调度性判定条件，已经在一般情况下使用问题窗口分析框架的难点。在后面的第 8.8 节中，将介绍若何对一般情况进行分析。

（a）只考虑处理器调度的情况

对于只考虑处理器调度（即假设总是有足够的缓存空间）的可调度性分析可以分为如下步骤：

1. 假设任务集τ在 M 个处理器核心调度，并假设每个任务的上A_i都为 0（因此任务不会因为缓存空间不足而被阻塞）。

2. 假设任务集τ不可调度，且J_k（τ_k的一个任务实例）为第一个不满足截止期的任务实例。J_k的释放时间为r_k。$l_k = r_k + S_k$为其能满足截止期时最晚的开始执行期间时间（实际上J_k未能在l_k时开始执行因为它错失截止期了）。定义长度为S_k的时间区域$[r_k, l_k]$为问题窗口，如图 8.3-(a)所示。问题窗口中所有的时间点上所有的处理器都必须被其它任务所占，这样才使J_k无法在问题窗口内开始执行而错失截止期。

3. 计算每个任务τ_i在问题窗口内所产生的工作量的上限值，也称为τ_i对J_k的干涉，记为I_k^i。所有任务干涉之和$\sum_i I_k^i$为问题窗口里J_k所承受的干涉的总和上限。下一章中将介绍如何计算这个上限。

4. 传统的非抢占固定优先级调度算法（不考虑缓存上的调度）具有如下性质：当有任务实例等待执行时，任意处理器核心都不可能空闲。因此，J_k只有在条件$\sum_i I_k^i \geq S_k \cdot M$满足的前提下才可能错失截止期，也就是说，只有图 8.3-(a)中所有阴影部分都被占满时，J_k才可能错失截止期。否则的话，如果满足条件$\sum_i I_k^i < S_k \cdot M$，则$J_k$一定是可调度的。

上述过程中的最后一步也可以通过另一种方式来理解。J_k所承受的干涉总和上限为$\sum_i I_k^i$，且在最坏情况下这些干涉是在 M 个处理器器核心上并行执行来阻止J_k执行的。因此，如果将$\sum_i I_k^i$除以 M，则可得的J_k被其它任务延迟的时间上限。称其为干涉时间。

如果J_k的干涉时间严格小于J_k的松弛时间，即满足条件

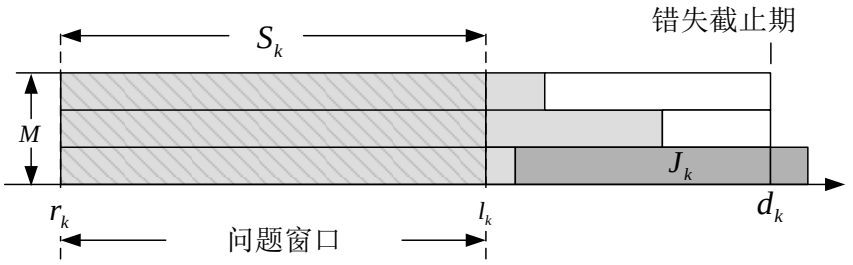
$$\frac{1}{M}\sum_i I_k^i < S_k \tag{8.1}$$

则J_k一定可调度。

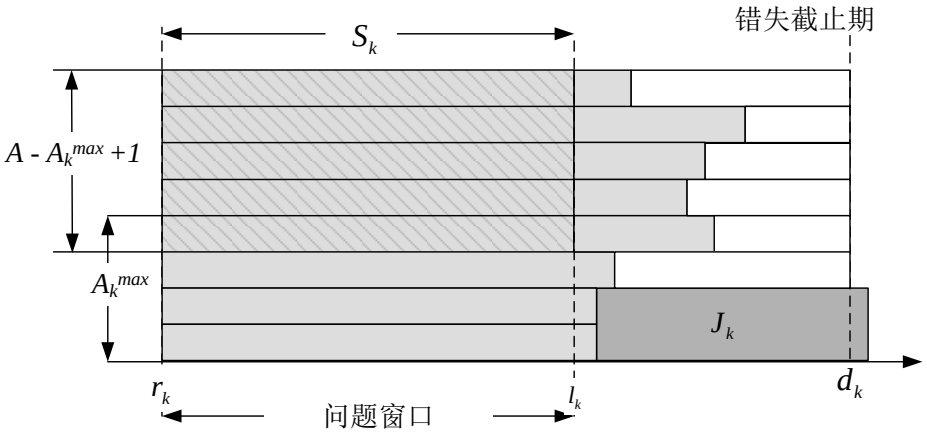
将上述过程应用于每个任务$\tau_k \in \tau$，（即检查上述条件是否对所有任务都满足），就可以为不考虑缓存调度的情况得到一个充分非必要的可调动行判断条件。

（b）只考虑缓存调度的情况

上述基于问题窗口的分析方法可以被扩展到每个任务需要若干个运算资源的情况。在这里，特指一个任务在运行时需要占用多个缓存块。



(a) 只考虑处理器核心调度
(a) Only consider scheduling on cores



(b) 只考虑缓存调度
(b) Only consider scheduling on cache

图 8.3 只考虑处理器核心调度或缓存调度的问题窗口示意图
Fig. 8.3 Illustration of problem window with scheduling only on cores or cache

假设只考虑针对缓存块的调度（假设总是具有足够处理器核心来执行任务）。一个任务实例 J_k 能够运行的条件是，其为等待队列 Q_{wait} 里的第一个任务实例，且目前空闲缓存块的个数至少为 A_k。如果目前空闲缓存块的数目少于 $\max(A_1,...,A_k)$，则等待队列 Q_{wait} 中的 J_k 有可能无法开始执行。注意，这里检查的所以高优先级任务所需要空闲缓存块数目的最大值 $\max(A_1,...,A_k)$，而不只是 A_k。这是因为，即使目前处理器上的空闲缓存块数不少于 A_k，依然可能因为有某个高优先级任务实例 J_i 无法得到足够的缓存块而无法开始执行，进而使 J_k 无法开始执行（因为 $\mathrm{FP}_{\mathrm{CA}}$ 为阻塞性调度算法）。定义如下概念：

$$A_k^{\max} = \max_{i \le k} A_i$$

$A_k^{\max}$ 是使 J_k 在任何情况下都不会被由于缓存块的原因而被阻塞无法执行所需要的最小的空闲缓存块数目。类似的，能够阻止 J_k 执行的被占用的缓存块的最小数目为 $A - A_k^{\max} + 1$。因此，只有当图 8.3-(b)的阴影部分全部被占满，J_k 才可能错失截止期。因

为每个任务 τ_i 在执行时占用 A_i 个缓存块，可知 J_k 只有在条件 $\sum_i A_i I_k^i \geq S_k \cdot \left(A - A_k^{\max} + 1\right)$ 被满足时才可能错失截止期。因此，可以得到只考虑缓存上调度的一个充分非必要的可调度性判定条件：

$$\sum_i A_i I_k^i < S_k \cdot \left(A - A_k^{\max} + 1\right)$$

与上一小节中一样，可以通过干涉时间的角度来理解上述条件：将 J_k 所承受的干涉的上限值除以 $\left(A - A_k^{\max} + 1\right)$，便可得到 J_k 能被其他任务延迟的时间上限。因此，得到可调度性判定条件：

$$\frac{1}{A - A_k^{\max} + 1}\left(\sum_i A_i I_k^i < S_k\right) \tag{8.2}$$

如条件(8.1)和(8.2)中所见，只要知道每个任务 τ_i 的 I_k^i，就可以为分别只考虑处理器核心和缓存块的调度算法建立可调度分析条件。然后，因为在调度算法 $\mathrm{FP}_{\mathrm{CA}}$ 中，调度同时发生在处理器核心与缓存块上，下两节将介绍如何将上述的两个条件通过某种方式结合起来为 $\mathrm{FP}_{\mathrm{CA}}$ 提供一个安全的可调度性分析条件。

8.5　基于线性规划问题求解的方法

为了将基于问题窗口的分析方法应用于 $\mathrm{FP}_{\mathrm{CA}}$，需要回答下述两个问题：

- 如何计算每个任务 τ_i 在问题窗口中的干涉上限 I_k^i？
- 如何决定所有任务的干涉之和是否足够阻止 J_k 在问题窗口中执行？

上面提出的第一个问题可以通过将问题窗口中的任务 τ_i 的实例 J_i 分为三类：（1）前部任务实例；（2）中部任务实例；（3）后部任务实例。因此可以得到 τ_i 在问题窗口里所产生的干涉时间为：

$$I_k^i = \left(\left\lfloor \frac{S_k}{T_i} \right\rfloor + 2\right) \cdot C_i \tag{8.3}$$

对于第二个问题的回答是本章所提出的分析方法的主要贡献。如第 8.4.2 节中介绍，问题窗口分析方法可以被单独应用于处理器核心或者缓存。但是，如果同时考虑处理器核心和缓存上的调度，则不知道在每种资源上能引起 J_k 错失截止期所需要的资源下限。例如，图 8.2-(b)中，在时刻 0，任务是实例 J_3^1 已经就绪，但是由于目前空闲的缓存块不足，它不能开始运行。因此，为了使被分析的任务错失截止期，不一定需要在问题窗口里所有的 M 个处理器核心都需要是忙碌的。

8.5.1　问题窗口的划分

本章提出的分析方法中的基本思路是，如果问题窗口中的某个时间点上，任务实例

不能开始执行，那么在此时间点上，要么所有的处理器核心都被占满了，要么缓存空间不足，两者至少满足一个。这可以概括为如下引理：

引理 8.1：令 J_k 为一个错失截止期的任务实例。在问题窗口内的任意一个时间点上，以下两个条件中至少有一个为真：

1. 所有 M 个处理器核心都被占用；

2. 至少 $A-A_k^{\max}+1$ 个缓存块被占用。

证明：用反证法证明。假设某个时间点 $t\in[r_k,l_k]$ 上引理中的两个条件都不满足。因为 J_k 错失截止期，其不能再问题窗口里开始执行，即其在 t 时刻没有执行。此外，Q_{wait} 在 t 时刻一定不为空，因为其至少包含 J_k。

令 J_i 为 t 时刻 Q_{wait} 中的第一个任务实例。因为 $\mathrm{FP_{CA}}$ 中等待队列是严格按照优先级排序的，J_i 的优先级一定是所有就绪任务实例中优先级最高的那个。根据加上，t 时刻被占用的缓存块至少为 $A-A_k^{\max}+1$，因此此刻可用的缓存块至少为 $A_k^{\max}$。因为 $A_i\le A_k^{\max}$，且根据假设此刻至少存在一个空闲的处理器核心，可知 J_i 在 t 时刻可以执行，这与 J_i 在 t 时刻处于 Q_{wait} 中等待相矛盾。　□

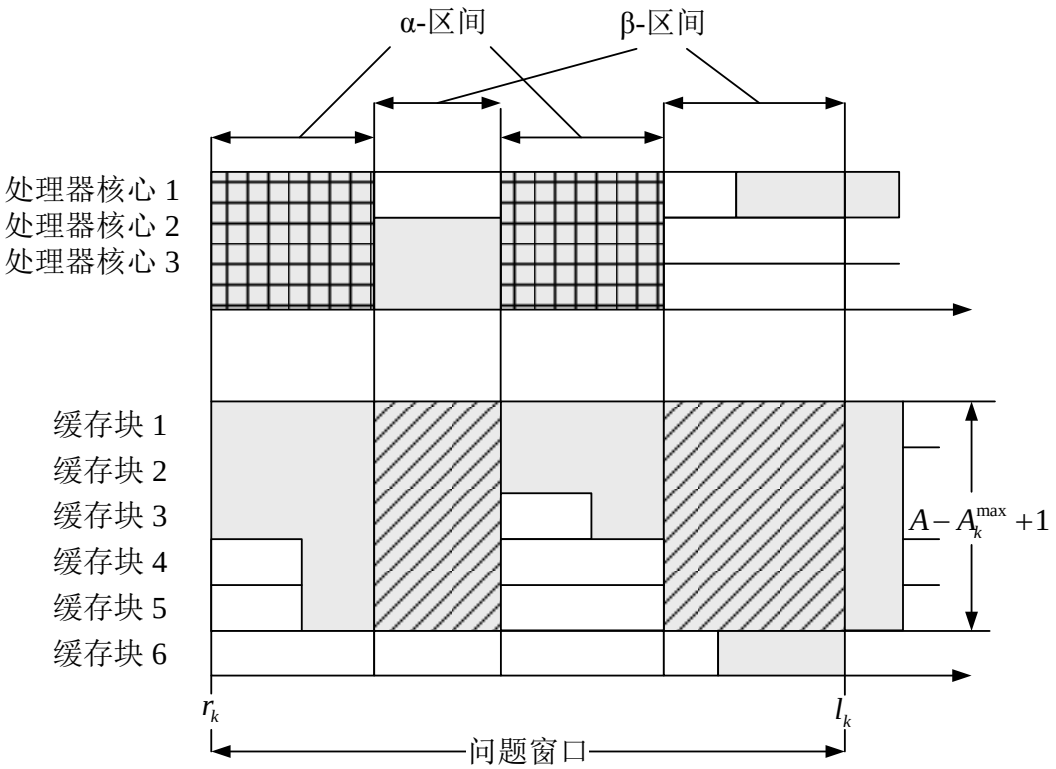


图 8.4 $\alpha-$区间和 $\beta-$示意图

Fig. 8.4 Illustration of $\alpha-$interval and $\beta-$interval

根据上述引理中的两个条件，可以将问题窗口划分为两部分（如图 8.4 所示）：

1. $\alpha-$区间：在 $\alpha-$区间内所有的处理器核心都被占用；

2. $\beta-$区间：在$\beta-$区间内至少有一个处理器核心为空闲。根据引理 8.1 可知，在$\beta-$区间内至少有$A-A_k^{\max}+1$个缓存块被正在运行的任务占用。

$\alpha-$区间或者$\beta-$区间未一系列满足相应定义的（非连续）时间区域的总称。一般来讲，并不知道在J_k承受最大干涉时$\alpha-$区间或者$\beta-$区间所保护时间区域长度的总和。下面，将介绍如何通过将这一问题转换为线性规划问题，进而建立$\mathrm{FP_{CA}}$的可调度性分析条件。

8.5.2 线性规划问题

和上文一样，假设一个任务集τ不可被$\mathrm{FP_{CA}}$调度，且J_k是其中第一个错失截止期的任务实例。时间区域$[r_k,l_k]$为问题窗口。

进行可调度性分析的线性规划问题涉及如下定义：

- M：处理器核心的数目
- A：共享缓存上缓存块的总数目
- A_i：每个任务τ_i运行时需要的缓存块的数目（还可以通过这些常数得到$A_k^{\max}$）
- I_k^i：每个任务τ_i在问题窗口内产生的干涉上限，这是根据上面介绍的方法所计算的。

此外，线性规划问题还使用如下非负变量：

- α_i:每个任务τ_i，使用α_i表示τ_i在$\alpha-$区间内累计执行的时间之和。
- β_i:每个任务τ_i，使用β_i表示τ_i在$\beta-$区间内累计执行的时间之和。

在$\alpha-$区间内，所有的 M 个处理器核心都被占用。此外，$\sum_i \alpha_i$为所有任务在$\alpha-$区间内执行时间总和（图 8.4 中方格阴影部分）。因此，可以得出$\alpha-$区间的长度总和为：

$$\frac{1}{M}\sum_i \alpha_i \tag{8.4}$$

在$\beta-$区间内，每一时刻至少$A-A_k^{\max}+1$个缓存块被占用。此外，$\sum_i A_i\beta_i$是所有任务在$\beta-$区间内占用的缓存块的总和（图 8.4 中斜线阴影部分）。因此，可以得出$\beta-$区间的长度总和为：

$$\frac{1}{A-A_k^{\max}+1}\sum_i A_i\beta_i \tag{8.5}$$

因为J_k不可调度，可知$\alpha-$区间内与$\beta-$区间长度的总和至少为S_k，因此，根据(8.4)和(8.5)可得：

$$\sum_i\left(\frac{1}{M}\alpha_i+\frac{A_i}{A-A_k^{\max}+1}\beta_i\right)\geq S_k \tag{8.6}$$

可以通过线性规划问题来判断是否可以找到一组α_i和β_i变量值来满足上述条件。如果不能，则可知τ_k一定是可以调度的。可以通过下述线性规划的目标函数：

$$Maximize \ \sum_i \left(\frac{1}{M}\alpha_i + \frac{A_i}{A - A_k^{\max} + 1}\beta_i \right) \tag{8.7}$$

如果线性规划问题的解小于S_k，则可以判定τ_k一定是可以调度的。

下面介绍线性规划问题的约束条件：

φ_1：干涉约束：因为每个任务τ_j在问题窗口内所执行工作量的上限为I_k^j，可知

$$\forall j : \alpha_j + \beta_j \le I_k^j$$

φ_2：处理器约束：一个任务在$\alpha-$区间内的执行时间不能超过$\alpha-$区间的总长度，因此有下列约束：

$$\forall j : \alpha_j \le \frac{1}{M}\sum_i \alpha_i$$

φ_3：缓存约束：一个任务在$\beta-$区间内的执行时间不能超过$\beta-$区间的总长度，因此有下列约束：

$$\forall j : \beta_j \le \frac{1}{A - A_k^{\max} + 1}\sum_i A_i\beta_i$$

通过求解上述有约束φ_1至φ_3和优化目标(8.7)组成的线性规划问题，可以对任务τ_k的可调度性进行判定，如下定理所述：

定理 8.1：对每个任务τ_k，令χ_k表示上述线性规划问题的解。如果任务集中τ每个任务τ_k都满足：

$$\chi_k < S_k \tag{8.8}$$

则任务集τ可以被$\mathrm{FP}_{\mathrm{CA}}$调度。

8.6　基于封闭表达式的方法

尽管上一节中基于线性规划问题求解的可调度性判定方法具有比较好的可延展性（如第 7 节所示），但是在很多情况下，比如作为运行时的准入控制或者在系统设计周期中的快速判断中，系统设计者可能会希望得到一个更简单的判定条件。因此，本节将介绍一个更加简单的判断条件，该条件是上一节中基于线性规划问题求解的判断的安全近似。本节的方法的计算复杂度为$O(N^2)$。

在基于线性规划问题求解的判断中，每个任务τ_i的干涉I_k^i被分为α_i和β_i两部分（如约束φ_1所示）。根据目标函数(8.7)可知如果满足：

$$1/M \geq A_i \Big/ \left(A - A_k^{\max} + 1\right)$$

则 τ_i 将尽可能地通过 α_i 来贡献其干涉（只要其满足约束 φ_2）。类似地，如果满足：

$$1/M < A_i \Big/ \left(A - A_k^{\max} + 1\right)$$

则 τ_i 将尽可能地通过 β_i 来贡献其干涉（只要其满足约束 φ_3）。然而，因为 $\alpha-$区间和 $\beta-$区间的长度（也就是约束 φ_2 和 φ_3 的右部）依赖于 α_i 和 β_i 这些变量，所有一般来讲无法知道如何取各个任务的 α_i 和 β_i 值来最大话所有任务的干涉时间。因此，上一节的判断条件通过将使用线性规划问题求解器来帮助“检测所有的可能性以获得最大目标值。

如果将约束 φ_2 与 φ_3 从线性规划问题中去除，为了最大化优化目标，对每个任务 τ_i，如果满足 $1/M \geq A_i \Big/ \left(A - A_k^{\max} + 1\right)$，则有 $\alpha_i = I_k^i$ 且 $\beta_i = 0$，否则则有 $\beta_i = I_k^i$ 和 $\alpha_i = 0$。因此，可以由此得到一个封闭表达式的可调度性分析判定条件，如下定理所述：

定理 8.2：对每个任务 τ_k，令

$$\chi_k^* := \sum_i \max\left(\frac{1}{M}, \frac{A_i}{A - A_k^{\max} + 1}\right) \cdot I_k^i$$

如果任务集中 τ 每个任务 τ_k 都满足：

$$\chi_k^* < S_k \tag{8.9}$$

则任务集 τ 可以被定理 8.1 判断为可被 $\mathrm{FP}_{\mathrm{CA}}$ 调度。

证明：通过反证法证明。假设 τ 为一个通过定理 1 判断为不可被 $\mathrm{FP}_{\mathrm{CA}}$ 调度的任务集，即定理 1 中的线性规划问题存在一个解满足不等式：

$$\sum_i \left(\frac{1}{M} \cdot \alpha_i + \frac{A_i}{A - A_k^{\max} + 1} \cdot \beta_i\right) \geq S_k$$

该不等式可被转化为

$$\sum_i \max\left(\frac{1}{M}, \frac{A_i}{A - A_k^{\max} + 1}\right) \cdot \left(\alpha_i + \beta_i\right) \geq S_k$$

应用约束 φ_1：

$$\underbrace{\sum_i \max\left(\frac{1}{M}, \frac{A_i}{A - A_k^{\max} + 1}\right) \cdot I_k^i}_{\chi_k^*} \geq S_k$$

定理得证。　□

M	$A-A_k^{\max}+1$	I_k^1	A_1	I_k^2	A_2	I_k^3	A_3
2	4	4	1	4	3	6	1

图 8.5 例子任务集的干涉与相关参数

Fig. 8.5 A task set example with related parameters and intererence

上述定理中得到的 χ_k^* 为前一节中线性规划问题解 χ_k 的一个安全近似。例如，假设一个任务集对某个任务的干涉以及相关参数如图 8.5 所示。则使用基于线性规划问题求解的可调度判断可以得到优化目标 χ_k=7，且各个变量的赋值为

$$\alpha_1=4 \quad \beta_1=0$$

$$\alpha_2=1 \quad \beta_2=3$$

$$\alpha_3=3 \quad \beta_3=3$$

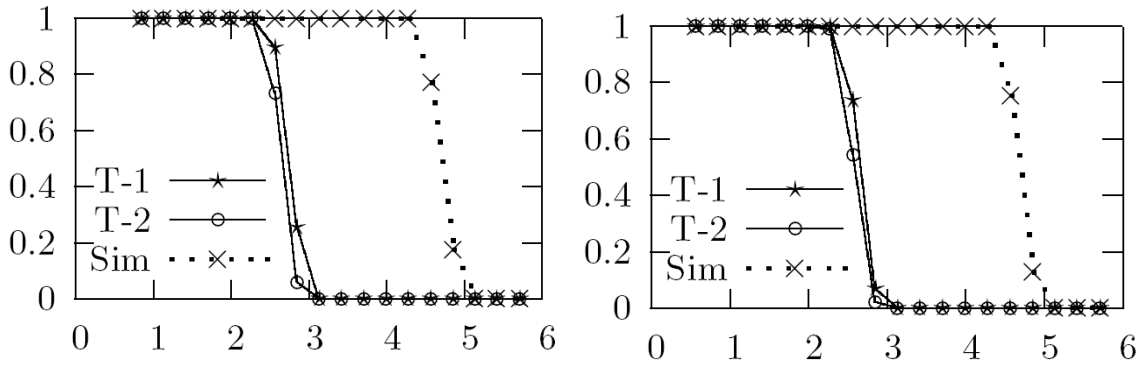
如果使用定理 8.2，则有

$$\chi_k^*=\frac{1}{2}\cdot 4+\frac{3}{4}\cdot 4+\frac{1}{2}\cdot 6=8$$

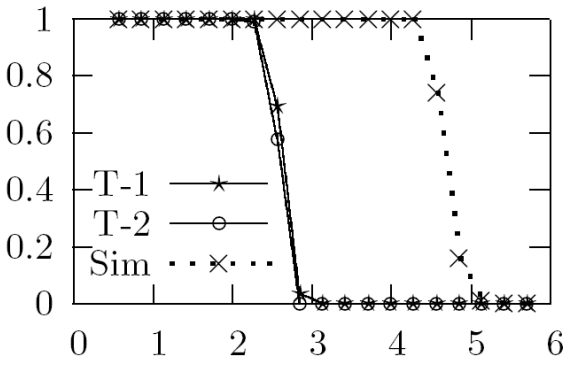
尽管理论上定理 8.2 比基于线性规划问题求解的判断要悲观一些，但是通过大量的实验表明，实际上定理 8.2 的性能（判定一个任务集可调度的能力）与基于线性规划问题求解的判断是非常接近的。这将在下一节中详细叙述。因此，对于实际系统来说，定理 8.2 的判断依然有较好的精确度，但是具有很低的计算复杂度。定理 8.1 与定理 8.2 的可调度性判定都是可持续的，也就是说，一个被判定为可调动的任务集，如果降低其负载（减小 C_i 或者增加 T_i），其一定仍被判定为可调度。由于篇幅的关系，这里将不给出详细证明。但是，定理 8.1 与定理 8.2 的可调度性判定对于相对截止期和执行时间并非是自可持续的。

8.7　性能评估

首先评估所提出的可调度性判定条件的精确性（接受率）。图 6 中所示为基于线性规划问题求解的判断（图中表示为“T-1”），基于封闭表达式的判断（图中表示为“T-2”）以及模拟实验（图中表示为“Sim”）的接受率。因为在模拟中尝试所有可能的任务释放偏移和释放间隔的计算量非常大，所以在模拟实验中将每个任务的释放偏移量设为 0，且每个任务都严格地周期性释放。模拟实验一直运行到所有任务周期的最小公倍数。



(a) $0.1 \le U_i \le 0.3,\ 1 \le A_i \le 5$　　(b) $0.1 \le U_i \le 0.6,\ 1 \le A_i \le 5$



(c) $0.1 \le U_i \le 0.3,\ 2 \le A_i \le 10$

图 8.6 接受率实验结果：X 轴为资源利用率总和，Y 轴为接受率

Fig. 8.6 Experiment results of acceptance ratio: X-axis is total utilization; Y-axis is acceptance ratio

图 8.6(a)中实验的参数设置如下：处理器核心的个数为 6，缓存块的个数为 40，每个任务 τ_i 的周期 T_i 均匀分布于[10,20]内，资源利用率 U_i 均匀分布在[0.1, 0.3]内，所需缓存块个数 A_i 均匀分布在[1, 5]内，且设置 $D_i = T_i$。可以看到，基于线性规划问题求解的判断的接受率略高于基于封闭表达式的判断。在图 8.6(b)中，资源利用率 U_i 的范围改为[0.1, 0.6]而保持其它设置不变。而图 8.6(c)中，A_i 的范围改为[2, 10]而保持其它设置不变。可以看到，在不同设置下，基于线性规划问题求解的判定的接受率和基于封闭表达式的判定的接受率都相当接近。

Number of Tasks	4000	6000	8000	10000
Time in LP(s)	49.24	114.53	208.45	334.95
Mem. in LP(KB)	20344	28876	37556	46664

图 8.6 可延展性实验结果

Fig. 8.6 Experiment results of scalability

如前文所述，基于封闭表达式的可调度性判定的时间复杂度为 $O(N^2)$。下面将重点评估基于线性规划问题求解的判定的效率。图 8.6 中所示为在求解不同规模任务集所需

要的时间和最大内存消耗量。实验环境为，使用开源的线性规划求解器 lpsolve[152]在一个具有 Intel Core2 处理器(2.83GHz)的桌面电脑进行求解。实验表明，对一个具有上千个任务的任务集的求解可以在几分钟内完成。

8.8 阻塞与非阻塞调度

如 4.1 节中所述，在某些情况下 FP_{CA} 可能引起资源浪费。这是由于任务必须严格按照优先级顺序开始执行造成的。比如，某一时刻空闲缓存块不够让等待队列里的第一个（即优先级最高的）任务开始运行，而此时等待队列里可能存在某个低优先级任务可以被放入目前空闲的缓存块里。在 FP_{CA} 中，这个低优先级任务不允许在其它等待的高优先级任务开始之前开始执行。这类调度策略称为阻塞调度策略。如上例所示，阻塞调度策略可能因为严格遵守任务的优先级顺序而造成资源浪费。但是，其好处是不会因为造成无限的优先级反转问题。

另一种选择是允许低优先级任务实例在等待队列的第一个任务实例开始之前开始执行。这样做可以某种程度上提供资源的利用率。这种调度策略成为非阻塞调度策略。此种调度下，调度器总是选择在所有实际上可以开始运行的任务实例中优先级最高的那个开始运行。例如，图 8.7 中所示为使用非阻塞版的 FP_{CA} 来调度图 8.2-(a)中的任务集。在 0 时刻，尽管任务实例 J_3^1 不能开始执行，调度器依然让 J_4^1 开始执行。

从系统可预测性的角度，通常阻塞调度更具有优势，因为其中任务实例严格遵守其优先级顺序开始运行。在非阻塞调度中，一个任务往往可能承受更多的干涉，这是因为一个低优先级任务可能在一个高优先级任务之前开始执行，而由于调度是非抢占的，这些开始了的低优先级任务必须执行至结束。如图 8.7 中所示，由于 J_4^1 的提前执行，J_3^1 的开始时间被推迟至时刻 3，而最终导致其错失截止期。在最坏情况下，这种优先级反转可能是无限的。

另一方面，非阻塞调度平均情况下具有更好的资源利用率，因为它总是尽可能地使用现有可用的资源。从运行时开销的角度，非阻塞调度需要查看等待队列第一个以外的任务实例是否可以开始运行。而阻塞调度则开销更小，因为它永远只需要关注等待队列中的第一个任务实例。

系统设计者可以根据应用的需求来选择阻塞调度或者非阻塞调度策略，甚至两种的混合方法。本章将不详细探讨和比较这些不同的可能性。但是，尽管本章介绍的可调度性分析方法是针对阻塞调度策略的，其也适用于非阻塞调度策略。对非阻塞调度策略进行分析，只需要将由可以提前执行的低优先级任务实例的干涉考虑到每个任务所承受的总干涉中去。

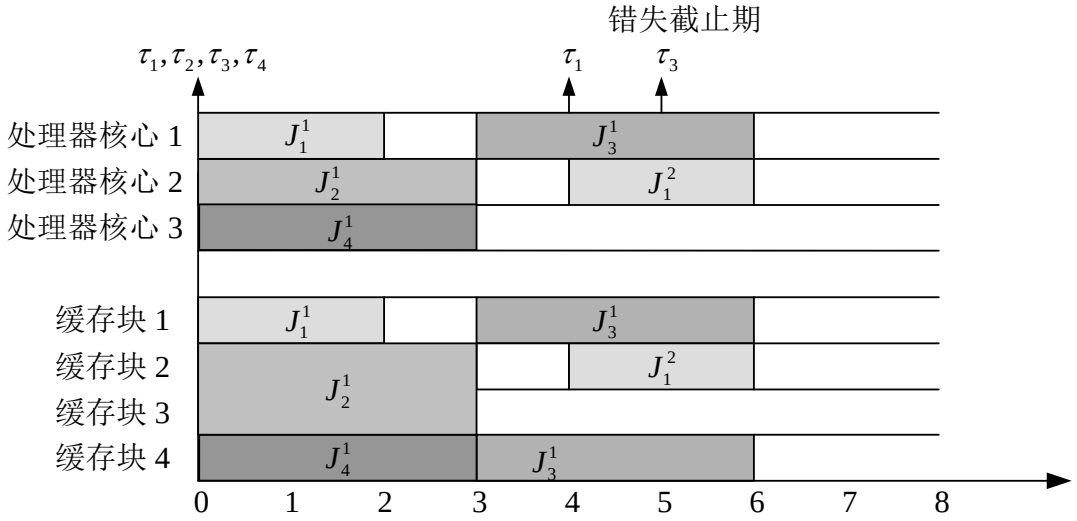


图 8.7 非阻塞的 $\mathrm{FP}_{\mathrm{CA}}$ 示意图

Fig. 8.7 Illustration of the non-blocking version of $\mathrm{FP}_{\mathrm{CA}}$

8.9 小结

多核处理器的飞速发展和广泛使用对嵌入式系统的设计带来了巨大的挑战。其中之一就是如何保证多核处理器上软件系统的实时性。片上共享资源，如共享缓存，会大大影响多处理机系统的时间可预测性。这一章工作主要传递的信息是，通过使用适合的资源隔离技术，可以通过使用现有的程序执行时间分析技术来保证系统的任务级别实时性，并最终对整个系统进行可调度性分析来保证整个系统的时间可以预测性。本章的主要技术贡献点为提出了一个基于全局固定优先级调度的缓存敏感调度算法及其两个可调度判定条件：第一个判断条件基于线性规划问题求解，第二个判定条件为封闭表达式。通过使用随机生成的实验表明，本章所提出的基于线性规划问题求解的可调度性判定方法具有较好的分析效率，而第二个基于本章提出的基于封闭表达式的可调度性判定方法在具有较低运算复杂度的同时，可以获得与第一个分析方法非常相近的分析质量。

第9章 结 论

随着多核处理器的飞速发展和广泛应用，越来越多的嵌入式实时系统将采用基于多核处理器的硬件平台。然而与硬件方面的飞速发展相比，面向多核系统的实时系统的软件设计与分析技术已经远远落后。虽然在过去四十多年里，面向单核处理器的实时调度理论已经取得了丰硕的理论成果并在实际应用中获得了巨大的成功，面向多核处理器实时调度方面的发展还远远没有成熟。本文围绕多核实时调度这一问题进行研究，在多处理机调度的关键时刻，有限响应时间，资源利用率界限等方面做出了重要理论贡献。

9.1 本文的主要贡献与结论

本文主要有 6 个贡献点：

（1）提出了一种全新的针对可抢占全局固定优先级调度的响应时间分析技术。该分析技术的特点是建立了一个与单处理机调度中的关键时刻类似的概念，通过构造一个相对具体的情形来代表系统的最坏情况行为。通过对这个特定情形进行分析，可以得到系统可调度性和响应时间的安全近似。在此基础上，建立了可抢占全局固定优先级调度下任务具有有限响应时间的一般性条件。

（2）提出了一种新的针对不可抢占全局固定优先级调度的可调度性分析技术。该分析技术的特点是为不可抢占固定优先级全局调度算法建立了一个新的“问题窗口”概念，并将上一部分中针对可抢占调度的分析技术应用于不可抢占调度。通过为不可抢占全局调度进行更加精确的分析以及进行大量的模拟实验，推翻了从单处理机实时调度中衍生出来并被普遍接受的关于可抢占调度的实时性能总是好于不可抢占调度的错误观念。本文对这种现象进行了深入分析，并讨论了在何种情况下可以更好地利用不可抢占调度来提高系统的实时性能。

（3）提出了一种固定实例优先级的全局调度算法及相应的分析技术。该调度算法与以往的全局调度算法有显著的区别：算法通过发掘任务实例之间优先级顺序来大幅度提高系统的可调度性。这种方法可以看作是将基于固定任务优先级分配技术与 EDF 这两种现有全局调度算法优点的结合。本文提出的算法通过构建运行时系统负载的抽象，只需在设计时对有限个具体的任务实例进行优先级分配，而在运行时系统将通过复用上述优先级分配方案来争正确进行调度。实验表明，该算法的性能明显优于基于固定任务优先级分配和 EDF 的全局调度算法。

（4）提出了一种新的准划分调度算法，将单处理机调度中著名的 Liu&Layland 资

源利用率界限扩展到多处理机调度模型。该算法具有多项式复杂度。其基本思想是使用与装箱问题中的“最坏适用递减”启发式算法类似的任务划分顺序，来使任务切割只发生在高优先级任务中，并利用高优先级任务具有较大松弛时间的特性，来抵消任务切割做带来的负载增长。这一算法的提出解决了实时调度领域一个近四十年悬而未决的重要理论问题。

（5）提出了一种新的准划分调度算法将单处理调度中大部分的参数化资源利用率界限扩展到了多处理机调度。在系统设计时，如果可以知道任务的更多参数，则有可能使用更高的参数化资源利用率界限来进行可调动行判断。提出的准划分调度算法可以将单处理上大部分的参数化资源利用率界限扩展到多处理机调度。该算法具有分析效率高（具有伪多项式复杂度）的优点外，由于使用了响应时间分析而非资源利用率界限来决定一个处理器上可以接纳的最大负载，因此获得了更好的平均实时性能。

（6）提出了一种新的共享缓存敏感的多核调度算法及相应的分析技术。本文提出一种基于共享缓存隔离的多核实时调度算法。该方法使用软件缓存划分技术，通过与调度算法的结合来隔离同时执行的硬实时任务的缓存空间，从而避免他们的相互干扰。该算法的基本思想是利用操作系统中虚拟地址与物理地址的映射，来将共享缓存划分成若干个缓存块。一个任务可以使用指定个数的缓存块，因此可以独立地使用现有基于单处理器系统的分析技术计算每个任务的最坏情况执行时间，进而保障整个系统的时间可预测性。

9.2 进一步的工作

本文进行的研究是针对多处理机实时调度研究中普遍采用的任务模型开展的。实际系统中的任务系统通常比本文采用周期性任务集更加灵活和复杂。此外，多核处理器硬件方面的飞速发展也将使实际系统所面临的调度问题更加复杂。基于已经获得的研究成果，在未来工作中，我们将在以下几个方面进一步深入研究：

（1）本文采用对等多核处理器模型，即各个处理器核心具有完全相同的处理能力。然而现在多核处理器系统一个重要的发展方向是采用异构的处理器结构，即在一个芯片上集成若干不同算能力或不同类型的处理器核心。因此，未来应继续开展对异构多核处理器上实时调度问题的研究。其中一个重要的问题是如何将 Liu&Layland 资源利用率推广到异构多核处理器调度。

（2）本文所采用的周期性任务模型假设各个任务间是完全相互独立的。在一些实际系统中各个任务之间并非完全独立的。例如，某些任务之间可能需要共享一些软件或硬件资源。对于此类系统，需要在针对抽象任务系统模型的实时调度算法之上使用资源

共享协议对任务的调度进行辅助管理。因此，未来应继续开展对多处理机调度中资源共享协议的研究。

（3）本文所采用的周期性任务模型假设各个任务的执行时完全串行的。实际系统中许多程序都可以被不同程度地并行化来进一步提高系统的资源利用效率。因此，未来应继续开展对多核处理器上并行程序任务模型的调度问题。其中一个重要的问题是如何将本文中建立的近似关键时刻的概念推广到并行实时任务模型。

（4）本文所研究的缓存敏感实时调度算法只考虑各个核上运行的任务由于共享缓存所造成的相互干扰，而没有考虑其他共享资源，如片上共享总线，共享存储器接口等所带来的影响。本文提出的共享缓存机制为空间隔离，而共享总线和共享存储器接口等资源只能进行时间隔离，因此对实时调度算法带来新需求有所不同。未来将继续开展共享总线敏感和共享存储器接口敏感的实时调度算法与分析问题，并最终将在实时调度中对各种共享资源的隔离与管理统一起来。

参考文献

1. Alan Burns and Andy Wellings. Real-Time Systems and Programming Languages (Third Edition) Ada 95, Real-Time Java and Real-Time POSIX [M], Addison Wesley Longmain, 2001.
2. Hermann Kopetz. Real-Time Systems, Design Principles for Distributed Embedded Applications [M], Kluwer Academic Publisshers, 1997.
3. Edward Lee. Cyber Physical Systems: Design Challenges [A], Proceedings of the IEEE Symposium on Object-Oriented Real-Time Distributed Computing [C], 2008, 363-369.
4. Jane. W.S. Liu. Real-Time Systems [M]. United States: Pearson Education, 2002.
5. Lui Sha, Tarek Abdelzaher, Karl-Erik Arzen, Anton Cervin, Theodore Baker, and Alan. Burns, Giorgio Buttazzo, Marco Caccamo, John Lehoczky, and Aloysius K. Mok, Real Time Scheduling Theory: A Historical Perspective [J], Journal of Real Time Systems, 2004, 28(2-3): 101-155.
6. www.arm.com/products/processors/classic/arm11/arm11-mpcore.php [EB/OL], 2012
7. www.arm.com/products/processors/cortex-a/cortex-a9.php [EB/OL], 2012
8. www.ti.com/general/docs/omap4 [EB/OL], 2012
9. www.ti.com/lsds/ti/dsp/platform/davinci [EB/OL], 2012
10. www.freescale.com/webapp/sps/site/homepage.jsp?code=POWERQUICC_HOME [EB/OL], 2012
11. researcher.watson.ibm.com/researcher/view_project.php?id=2649 [EB/OL], 2012
12. www.nxp.com/products/audio_video/nexperia/ [EB/OL], 2012
13. www.st.com/Nomadik [EB/OL], 2012
14. C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment [J]. In Journal of the ACM, 1973, 20(1): 46-61.
15. Sudarshan K. Dhall and C. L. Liu. On a real-time scheduling problem [J]. In Operations Research, 1978, 26(1): 127-140.
16. C. L. Liu. Scheduling algorithms for multiprocessors in a hard real-time environment. In JPL Space Programs Summary, 1969.
17. Michael R. Garey and David S. Johnson. Computers and Intractability: a Guide to the Theory of NP-Completeness [M]. W. H. Freeman and company, NY, 1979.

18. Yingfeng Oh and Sang H. Son. Allocating Fixed Priority Periodic Tasks on Multiprocessor Systems [J]. Real Time Systems, 1995, 9(3):207–239.
19. Almut Burchard, Jörg Liebeherr , Yingfeng Oh , Sang H. Son. New Strategies for Assigning Real-Time Tasks to Multiprocessor Systems [J], IEEE Transactions on Computers, 1995, 44(12).
20. Jose Maria Lopez, Manuel Barrena Garcia, Jose Luis Diaz, and Daniel Fernando Garcia. Worst-case utilization bound for EDF scheduling on real-time multiprocessor systems [A], Proceedings of the Euromicro Conference on Real-time Systems [C], 2000, 25-33.
21. Jose Maria Lopez, Jose Luis Diaz, Manuel Barrena Garcia, and Daniel Fernando Garcia. Utilization bounds for multiprocessor RM scheduling [J], Real-Time Systems, 2003, 24(1): 5–28.
22. Jose Maria Lopez, Jose Luis Diaz, and Daniel Fernando Garcia. Minimum and Maximum Utilization Bounds for Multiprocessor Rate Monotonic Scheduling [J], IEEE Transactions on Parallel and Distributed Systems, 2004, 15(7): 642-653.
23. Jose Maria Lopez, Manuel Barrena Garcia, Jose Luis Diaz, and Daniel Fernando Garcia. Utilization Bounds for EDF scheduling on Real-Time Multiprocessor Systems [J], Real Time Systems, 2004, 28(1): 39-68.
24. Björn Andersson, Sanjoy Baruah, and Jan Jonsson. Static-priority scheduling on multiprocessors [A]. Proceedings of the 22nd IEEE Real-Time Systems Symposium [C], 2001, 193–202.
25. Björn Andersson, and Jan Jonsson. The Utilization Bounds of Partitioned and Pfair Static-Priority Scheduling on Multiprocessors are 50% [A]. Proceedings of the 15th Euromicro Conference on Real-TimeSystems [C], 2003.
26. Cynthia A. Phillips, Cliff Stein, Eric Torng, and Joel Wein. Optimal time-critical scheduling via resource augmentation [A]. Proceedings of the ACM Symposium on theory of Computing [C], 1997, 140-149.
27. Björn Andersson, Sanjoy Baruah, and Jan Jonsson. Static-Priority Scheduling on Multiprocessors [A]. Proceedings of the IEEE Real-Time Systems Symposium [C], 2001, 193-202.
28. Theodore P. Baker. Multiprocessor EDF and Deadline Monotonic Schedulability Analysis [A]. Proceedings of the IEEE Real-Time Systems Symposium [C], 2003, 120-129.
29. Theodore P. Baker. An Analysis of EDF Schedulability on a Multiprocessor [J]. IEEE

Transactions on Parallel and Distributed Systems, 2005, 16(8): 760-768.

30. Theodore P. Baker. An Analysis of Fixed-Priority Schedulability on a Multiprocessor [J]. Real-Time Systems 32(1-2): 49-71 (2006)
31. Theodore P. Baker and Michele Cirinei. A Necessary and Sometimes Sufficient Condition for the Feasibility of Sets of Sporadic Hard-Deadline Tasks [A]. Proceedings of the IEEE Real-Time Systems Symposium [C], 2006, 178-190.
32. Theodore P. Baker and Sanjoy Baruah. An analysis of global edf schedulability for arbitrary-deadline sporadic task systems [J]. Real-Time Systems, 2009, 43(1): 3-24.
33. Theodore P. Baker and Sanjoy Baruah. Sustainable Multiprocessor Scheduling of Sporadic Task Systems [A]. Proceedings of the Euromicro Conference on Real-time Systems [C], 2009, 141-150.
34. Sanjoy Baruah. Techniques for Multiprocessor Global Schedulability Analysis [A]. Proceedings of the IEEE Real-Time Systems Symposium [C], 2007, 119-128.
35. Sanjoy Baruah and Nathan Fisher. Global Deadline-Monotonic Scheduling of Arbitrary-Deadline Sporadic Task Systems [A]. Procedings of the International Conference on Principles of Distributed Systems [C], 2007, 204-216.
36. Sanjoy Baruah and Theodore P. Baker. Schedulability analysis of global edf. Real-Time Systems 38(3): 223-235 (2008)
37. Sanjoy Baruah and Theodore P. Baker. Global EDF Schedulability Analysis of Arbitrary Sporadic Task Systems [A]. Proceedings of the Euromicro Conference on Real-time Systems [C], 2008, 3-12.
38. Sanjoy Baruah, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, and Sebastian Stiller. Improved multiprocessor global schedulability analysis [J]. Real-Time Systems, 2010, 46(1): 3-24.
39. Marko Bertogna, Michele Cirinei, Giuseppe Lipari: New Schedulability Tests for Real-Time Task Sets Scheduled by Deadline Monotonic on Multiprocessors [A]. Procedings of the International Conference on Principles of Distributed Systems [C], 2005, 306-321.
40. Marko Bertogna, Michele Cirinei, Giuseppe Lipari. Improved Schedulability Analysis of EDF on Multiprocessor Platforms [A]. Proceedings of the Euromicro Conference on Real-time Systems [C], 2005, 209-218.
41. Marko Bertogna, Michele Cirinei. Response-Time Analysis for Globally Scheduled

Symmetric Multiprocessor Platforms [A]. Proceedings of the IEEE Real-Time Systems Symposium [C], 2007, 149-160.

42. Marko Bertogna, Michele Cirinei, Giuseppe Lipari. Schedulability Analysis of Global Scheduling Algorithms on Multiprocessor Platforms [J]. IEEE Transactions on Parallel and Distributed Systems, 2009, 20(4): 553-566.
43. Marko Bertogna, Sanjoy Baruah. Tests for global EDF schedulability analysis [J]. Journal of Systems Architecture - Embedded Systems Design, 2011, 57(5): 487-497.
44. Björn Andersson, Eduardo Tovar. Multiprocessor Scheduling with Few Preemptions [A]. Proceedings of the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications [C], 2006, 322-334.
45. Björn Andersson, Konstantinos Bletsas. Sporadic Multiprocessor Scheduling with Few Preemptions [A]. Proceedings of the Euromicro Conference on Real-time Systems [C], 2008, 243-252.
46. Björn Andersson, Konstantinos Bletsas, Sanjoy. Baruah. Scheduling Arbitrary-Deadline Sporadic Task Systems on Multiprocessors [A]. Proceedings of the IEEE Real-Time Systems Symposium [C], 2009, 385-394.
47. Konstantinos Bletsas, Björn Andersson. Preemption-Light Multiprocessor Scheduling of Sporadic Tasks with High Utilisation Bound [A]. Proceedings of the IEEE Real-Time Systems Symposium [C], 2009, 447-456.
48. Konstantinos Bletsas, Björn Andersson. Notional Processors: An Approach for Multiprocessor Scheduling [A]. Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium [C], 2009, 3-12.
49. Shinpei Kato, Nobuyuki Yamasaki, Yutaka Ishikawa. Semi-partitioned Scheduling of Sporadic Task Systems on Multiprocessors [A]. Proceedings of the Euromicro Conference on Real-time Systems [C], 2009, 249-258.
50. Shinpei Kato, Nobuyuki Yamasaki. Semi-partitioned Fixed-Priority Scheduling on Multiprocessors [A]. Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium [C], 2009, 23-32.
51. Shinpei Kato, Nobuyuki Yamasaki. Portioned EDF-based scheduling on multiprocessors [A]. Proceedings of the IEEE International Conference on Embedded Software [C], 2008, 139-148.
52. Shinpei Kato, Nobuyuki Yamasaki. Portioned static-priority scheduling on

multiprocessors [A]. Proceedings of the IEEE International Symposium on Parallel and Distributed Processing [C], 2008, 1-12.

53. Shinpei Kato, Nobuyuki Yamasaki. Real-Time Scheduling with Task Splitting on Multiprocessors [A]. Proceedings of the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications [C], 2007, 441-450.
54. Karthik Lakshmanan, Ragunathan Rajkumar, John P. Lehoczky. Partitioned Fixed-Priority Preemptive Scheduling for Multi-core Processors [A]. Proceedings of the Euromicro Conference on Real-time Systems [C], 2009, 239-248.
55. Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, and et al. The Worst-Case Execution Time Problem - Overview of Methods and Survey of Tools [J], ACM Transaction on Embedded Computing Systems, 2008, 7(3): 1-53.
56. Andreas Schranzhofer, Rodolfo Pellizzoni, Jian-Jia Chen, Lothar Thiele, Marco Caccamo. Timing Analysis for Resource Access Interference on Adaptive Resource Arbiters [A]. Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium [C], 2011, 213-222.
57. Andreas Schranzhofer, Rodolfo Pellizzoni, Jian-Jia Chen, Lothar Thiele, Marco Caccamo. Worst-case response time analysis of resource access models in multi-core systems [A]. Proceedings of the Design Automation Conference [C], 2010, 332-337.
58. Andreas Schranzhofer, Jian-Jia Chen, Lothar Thiele. Timing Analysis for TDMA Arbitration in Resource Sharing Systems [A]. Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium [C], 2010, 215-224.
59. Simon Schliecker, Mircea Negrean, Rolf Ernst: Bounding the shared resource load for the performance analysis of multiprocessor systems [A]. Proceedings of the Design, Automation and Test in Europe Conference and Exposition [C], 2010, 759-764.
60. Simon Schliecker, Rolf Ernst. Real-time performance analysis of multiprocessor systems with shared memory [J]. ACM Transaction on Embedded Computer Systems, 2010, 10(2): 22.
61. Simon Schliecker, Mircea Negrean, Rolf Ernst. Response Time Analysis in Multicore ECUs with Shared Resources [J]. IEEE Transaction on Industrial Informatics, 2009, 5(4): 402-413.
62. John M. Calandrino, James H. Anderson. On the Design and Implementation of a Cache-Aware Multicore Real-Time Scheduler [A]. Proceedings of the Euromicro

Conference on Real-time Systems [C], 2009, 194-204.

63. John M. Calandrino, James H. Anderson: Cache-Aware Real-Time Scheduling on Multicore Platforms: Heuristics and a Case Study [A]. Proceedings of the Euromicro Conference on Real-time Systems [C], 2008, 299-308.
64. http://rtems.org/ [EB/OL], 2012.
65. http://www.litmus-rt.org/ [EB/OL], 2012.
66. www.qnx.com/ [EB/OL], 2012.
67. http://windriver.com/products/vxworks/ [EB/OL], 2012.
68. Yi Zhang, Nan Guan, Wang Yi and Yanbin Xiao. Implementation and Empirical Comparison of Partitioning-based Multi-core Scheduling [A], Proceedings of the IEEE International Symposium on Industrial Embedded Systems [C], 2011.
69. E. G. Coffman, Michael Randolph Garey, and D. S. Johnson. Approximation algorithms for bin packing: a survey. Approximation algorithms for NP-hard problems [M], 1997, 46 – 93.
70. Sanjoy Baruah, Alan Burns. Sustainable Scheduling Analysis [A]. Proceedings of the IEEE Real-Time Systems Symposium [C], 2006, 159-168.
71. Theodore P. Baker, Sanjoy Baruah. Sustainable Multiprocessor Scheduling of Sporadic Task Systems [A]. Proceedings of the Euromicro Conference on Real-time Systems [C], 2009, 141-150.
72. Theodore P. Baker. A comparison of global and partitioned edf schedulability tests for multiprocessors [A]. In Technical Report, Department of Computer Science, Florida State University [C], 2005.
73. Sanjoy Baruah, N. K. Cohen, C. Greg Plaxton, Donald A. Varvel. Proportionate progress: A notion of fairness in resource allocation [J]. Algorithmica, 1996, 600-625.
74. James H. Anderson and Anand Srinivasan. Mixed pfair/erfair scheduling of asynchronous periodic tasks [A], Proceedings of the Euromicro Conference on Real-time Systems [C], 2001, 76-88.
75. James H. Anderson, Anand Srinivasan. Mixed Pfair/ERfair scheduling of asynchronous periodic tasks [J]. Journal of Computer and Systems Science, 2004, 68(1), 157-204.
76. Anand Srinivasan, Philip Holman, James H. Anderson, Sanjoy Baruah: The Case for Fair Multiprocessor Scheduling. IEEE International Symposium on Parallel and Distributed Processing 2003: 114.

77. James H. Anderson, Aaron Block, Anand Srinivasan. Quick-release Fair Scheduling [A]. Proceedings of the IEEE Real-Time Systems Symposium [C], 2003, 130-141.
78. Anand Srinivasan, James H. Anderson. Optimal Rate-based Scheduling on Multiprocessors [A]. Proceedings on Annual ACM Symposium on Theory of Computing [C], 2002, 189-198.
79. James H. Anderson, Anand Srinivasan. Early-release fair scheduling [A]. Proceedings of the Euromicro Conference on Real-time Systems [C], 2000, 35-43.
80. James H. Anderson, Anand Srinivasan. Pfair scheduling: beyond periodic task systems [A]. Proceedings of the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications [C], 2000, 297-306.
81. Dakai Zhu, Daniel Mosse, and Rami Melhem. Multiple-resource periodic scheduling problem: how much fairness is necessary [A]. Proceedings of the IEEE Real-Time Systems Symposium [C], 2003, 142-151.
82. Hyeonjoong Cho, Binoy Ravindran, and E.Douglas Jensen. An optimal realtime scheduling algorithm for multiprocessors [A]. Proceedings of the IEEE Real-Time Systems Symposium [C], 2006, 101-110.
83. Shelby Funk. Lre-tl: an optimal multiprocessor algorithm for sporadic task sets with unconstrained deadlines [J]. Real-Time Systems, 2010, 46(3): 332-359.
84. Greg Levin, Shelby Funk, Caitlin Sadowski, Ian Pye, and Scott Brandt. Dpfair: A simple model for understanding optimal multiprocessor scheduling [A]. Proceedings of the Euromicro Conference on Real-time Systems [C], 2010, 3-13.
85. Ernesto Massa Greg Levin Paul Regnier, George Lima and Scott Brandt. Run: optimal multiprocessor real-time scheduling via reduction to uniprocessor [A]. Proceedings of the IEEE Real-Time Systems Symposium [C], 2011, 104-115.
86. Anand Srinivasan and Sanjoy Baruah. Deadline-based scheduling of periodic task systems on multiprocessors [J]. In Information Processing Letters, 2002, 84(2): 93-98.
87. Joël Goossens, Shelby Funk, and Sanjoy Baruah. Priority-driven scheduling of periodic task systems on multiprocessors [J]. Real-Time Systems, 2003, 25(2-3): 187-205.
88. Theodore P. Baker and Sanjoy Baruah. Schedulability analysis of global edf [J]. In Real-Time Systems, 2008, 38(3): 223-235.
89. Sanjoy Baruah, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, and Sebastian Stiller. Implementation of a speedup-optimal global edf schedulability test [A]. Proceedings of

the Euromicro Conference on Real-time Systems [C], 2009, 259-268.

90. Rami Melhem, Sylvain Lauzac and Daniel Mosse. Comparison of global and partitioning schemes for scheduling rate monotonic tasks on a multiprocessor [A]. Proceedings of the Euromicro Conference on Real-time Systems [C], 1998, 188-195.
91. Lars. Lundberg. Multiprocessor scheduling of age constraint processes [A]. Proceedings of the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications [C], 1998, 42-50.
92. Nan Guan, Zonghua Gu, Qingxu Deng, Shuaihong Gao, and Ge Yu. Exact schedulability analysis for static-priority global multiprocessor scheduling using model-checking [A]. Proceedings of Software Technologies for Embedded and Ubiquitous Systems [C], 2007, 263-272.
93. Theodore P. Baker and Michele Cirinei. Brute-force determination of multiprocessor schedulability for sets of sporadic hard-deadline tasks [A]. In Procedings of the International Conference on Principles of Distributed Systems [C], 2007, 62-75.
94. Nan Guan, Zonghua Gu, Mingsong Lv, Qingxu Deng, and Ge Yu. Exact schedulability analysis of global scheduling on multiprocessor platforms by symbolic model checking [A]. Proceedings of the IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing [C], 2008, 556-560.
95. UmaMaheswari C. Devi, James H. Anderson. Tardiness Bounds under Global EDF Scheduling on a Multiprocessor [A]. Proceedings of the IEEE Real-Time Systems Symposium [C], 2005, 330-341.
96. Robert I. Davis, Alan Burns. Priority Assignment for Global Fixed Priority Pre-Emptive Scheduling in Multiprocessor Real-Time Systems [A]. Proceedings of the IEEE Real-Time Systems Symposium [C], 2009, 398-409.
97. Robert I. Davis, Alan Burns. Improved priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems [J]. Real-Time Systems, 2011, 47(1): 1-40.
98. Seongie Cho, Suk-kyoon Lee, A. Han, and Kwei-Jay Lin. Efficient real-time scheduling algorithms for multiprocessor systems [J]. In IEICE Trans. Communication, 2002
99. Michele Cirinei and Theodore P. Baker. Edzl scheduling analysis [A]. In Proceedings of the Euromicro Conference on Real-time Systems [C], 2007, 9-18.

100. Shinpei Kato and Nobuyuki Yamasaki. Global edf-based scheduling with efficient priority promotion [A]. Proceedings of the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications [C], 2008, 197-206.
101. Robert I. Davis and Alan Burns. Fpzl schedulability analysis [A]. Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium [C], 2011, 245-256.
102. Jinkyu Lee, Arvind Easwaran, and Insik Shin. Maximizing contention-free executions in multiprocessor scheduling [A]. Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium [C], 2011, 235-244.
103. Björn Andersson, Sanjoy Baruah, and Jan Jonsson, 2001. Static-priority scheduling on multiprocessors [A]. Proceedings of the IEEE Real-Time Systems Symposium [C], 2001, 193-202.
104. Almut Burchard, Jörg Liebeherr , Yingfeng Oh and Sang H. Son. New Strategies for Assigning Real-Time Tasks to Multiprocessor Systems [J], IEEE Transactions on Computers, 1995, 44(12): 1429-1442.
105. Dong-IK Oh, Theodore P. Baker. Utilization bounds for N-processor rate monotone scheduling with stable processor assignment [J]. Real Time Systems, 1998, 15(2):183–193.
106. Jose Maria Lopez, Jose Luis Diaz, Manuel Barrena Garcia, and Daniel Fernando Garcia. Utilization bounds for multiprocessor RM scheduling [J]. Real-Time Systems 2003, 24(1):5–28.
107. Jose Maria Lopez, Jose Luis Diaz, and Daniel Fernando Garcia. Minimum and Maximum Utilization Bounds for Multiprocessor Rate Monotonic Scheduling, IEEE Transactions on Parallel and Distributed Systems, 2004, 15(7).
108. Björn Andersson, and Jan Jonsson, 2003. The Utilization Bounds of Partitioned and Pfair Static-Priority Scheduling on Multiprocessors are 50% [A]. Proceedings of the Euromicro Conference on Real-time Systems [C], 2003, 33-40.
109. Jose Maria Lopez, Manuel Barrena Garcia, Jose Luis Diaz, and Daniel Fernando Garcia. Worst-case utilization bound for EDF scheduling on real-time multiprocessor systems [A]. Proceedings of the Euromicro Conference on Real-time Systems [C], 2000, 25–33.
110. Jose Maria Lopez, Manuel Barrena Garcia, Jose Luis Diaz, and Daniel Fernando Garcia. Utilization Bounds for EDF scheduling on Real-Time Multiprocessor Systems [J], Real

Time Systems, 2004, 28(1): 39-68.

111. Neil C. Audsley, Alan. Burns, M. F. Richardson, Ken Tindell, and Andy J. Wellings. Applying new scheduling theory to static priority preemptive scheduling [J]. In Software Engineering Journal, 1993.
112. John P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines [A]. Proceedings of the IEEE Real-Time Systems Symposium [C], 1990, 201-209.
113. M. Gonzalez Harbour, J. Palencia Gutierrez. Schedulability analysis for tasks with static and dynamic offsets [A], Proceedings of the IEEE Real-Time Systems Symposium [C], 1998, 26-37.
114. Ken. Tindell, Hans. Hansson, and Andy. Wellings. Analysing realtime communications: Controller area network (can) [A]. Proceedings of the IEEE Real-Time Systems Symposium [C], 1994, 259-263.
115. Robert I. Davis, Alan Burns, Reinder J. Bril, Johan J. Lukkien. Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised [J]. Real-Time Systems, 2007, 35(3): 239-272.
116. Mathai Joseph, Paritosh K. Pandya. Finding Response Times in a Real-Time System [J]. The Computer Journal, 1986, 29(5): 390-395.
117. Björn Andersson and Jan Jonsson. Some insights on fixed-priority preemptive non-partitioned multiprocessor scheduling [A]. Technical Report, Chalmers University of Technology [C], 2001.
118. Alan. Burns and Andy Wellings. Real-time systems and programming languages [M]. Addison-Wesley, 3rd edition, 2001
119. Lars Lundberg. Multiprocessor scheduling of age constraint processes [A]. Proceedings of the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications [C], 1998. 42-50.
120. Manuel Blum, Robert W. Floyd, Vaughan Partt, Ronald L.Rivest, Robert E.Tarjan. Time bounds for selection [J]. Journal of Computer and System Sciences, 1973, 7(4), 448–461.
121. Theodore P. Baker and Michele Cirinei. A unified analysis of global edf and fixed task-priority schedulability of sporadic task systems on multiprocessors [J]. In Journal of Embedded Computing, 2011, 4(2): 55-69.
122. Sanjoy Baruah and Nathan Fisher. Global fixed-priority scheduling of arbitrary-deadline sporadic task system [A]. Proceedings of International Conference on Distributed

Computing and Networking [C], 2008, 215-226.

123. Kevin Jeffay, Donald F. Stanat. On non-preemptive scheduling of periodic and sporadic tasks [A]. Proceedings of the IEEE Real-Time Systems Symposium [C], 1991.
124. Laurent George, Nicolas Rivierre, Marco Spuri. Preemptive and non-preemptive real-time uni-processor scheduling [A]. Technical Report, INRIA [C], 1996.
125. Sanjoy Baruah, Samarjit Chakraborty. Schedulability analysis of non-preemptive recurring real-time tasks [A]. Proceedings of the IEEE International Symposium on Parallel and Distributed Processing [C], 2006, 1-8.
126. Sanjoy Baruah. The non-preemptive scheduling of periodic tasks upon multiprocessors [J]. Real-Time Systems, 2006, 32(1-2): 9-20.
127. Hennadiy Leontyev and James H. Anderson. A unified hard/soft real-time schedulability test for global edf multiprocessor scheduling [A]. Proceedings of the IEEE Real-Time Systems Symposium [C], 2008.
128. Neil C. Audsley. Flexible scheduling in hard-real-time systems [A]. In PhD thesis, Department of Computer Science, University of York, 1993.
129. Sanjoy Baruah, Aloysius K. Mok, Louis E. Rosier. Preemptively scheduling hard real-time sporadic tasks on one processor [A]. Proceedings of the IEEE Real-Time Systems Symposium [C], 1990, 182-190.
130. Tei Wei Kuo and Aloysius K. Mok. Load adjustment in adaptive real-time systems [A]. In Proceedings of the IEEE Real-Time Systems Symposium [C], 1991, 160-170.
131. Sylvain Lauzac, Rami Melhem, Daniel Mosse. An efficient rms admission control and its application to multiprocessor scheduling [A]. IEEE International Symposium on Parallel and Distributed Processing [C], 1998, 511-518.
132. Jun Yan and Wei Zhang. Wcet analysis for multi-core processors with shared l2 instruction caches [A]. Proceedings of the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications [C], 2008, 80-89.
133. Yan Li, Vivy Suhendra, Yun Liang, Tulika Mitra, Abhik Roychoudhury: Timing Analysis of Concurrent Programs Running on Shared Cache Multi-Cores [A]. Proceedings of the IEEE Real-Time Systems Symposium [C], 2009, 57-67.
134. Brian N. Bershad, J.Bradley Chen, Dennis Lee, and Theodore H. Romer. Avoiding conflict misses dynamically in large direct mapped caches [A]. Proceedings of the International Conference on Architectural Support for Programming Languages and

Operating Systems [C], 1994, 158-170.

135. Marco Paolieri, Eduardo Quiñones, Francisco J. Cazorla, Guillem Bernat, and Mateo Valero. Hardware support for wcet analysis of hard real-time multicore systems [A]. Proceedings of the International Symposium on Computer Architecture [C], 2009, 57-68.
136. Jakob Rosen, Alexandru Andrei, Petru Eles, Zebo Peng. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip [A]. Proceedings of the IEEE Real-Time Systems Symposium [C], 2007, 49-60.
137. Alexandra Fedorova, Margo Seltzer, Christopher Small and Daniel Nussbaum. Throughput-oriented scheduling on chip multithreading systems [A]. Technical Report, Harvard University [C], 2005.
138. Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. Predicting inter-thread cache contention on a multi-processor architecture [A]. Proceedings of the International Conference on High-Performance Computer Architecture [C], 2005, 340-351.
139. James H. Anderson and John M. Calandrino. Parallel real-time task scheduling on multicore platforms [A]. Proceedings of the IEEE Real-Time Systems Symposium [C], 2006, 89-100.
140. James H. Anderson, John M. Calandrino, and UmaMaheswari C. Devi. Real-time scheduling on multicore platforms [A]. Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium [C], 2006, 179-190.
141. Klaus Danne and Marco Platzner. An edf schedulability test for periodic tasks on reconfigurable hardware devices [A]. Proceedings of the ACM Conference on Languages, Compilers, and Tools for Embedded Systems [C], 2006, 93-102.
142. Nan Guan, Qingxu Deng, Zhonghua Gu, Wenyao Xu, and Ge Yu. Schedulability analysis of preemptive and non-preemptive edf on partial runtime-reconfigurable fpgas [J]. In ACM Transaction on Design Automation of Electronic Systems, 2008, 13(4).
143. Nathan Fisher, James H. Anderson, and Sanjoy Baruah. Task partitioning upon memory-constrained multiprocessors [A]. Proceedings of the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications [C], 2005, 416-421.
144. Vivy Suhendra, Chandrashekar Raghavan, and Tulika Mitra. Integrated scratchpad memory optimization and task scheduling for mpsoc architectures [A]. Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded

Systems [C], 2006, 401-410.

145. Hassan Salamy and Jagannathan Ramanujam. A framework for task scheduling and memory partitioning for multi-processor system-on-chip [A]. Proceedings of the International Conference on High Performance Embedded Architectures and Compilers [C], 2009, 263-277.
146. Vivy Suhendra and Tulika Mitra. Exploring locking and partitioning for predictable shared caches on multi-cores. In DAC, 2008.
147. Derek Chiou, Rinivas Devadas, Larry Rudolph, and Boon S. Ang. Dynamic cache partitioning via columnization [A]. In Technical Report, MIT [C], 1999.
148. Andrew Wolfe. Software-based cache partitioning for real-time applications [J]. In Journal of Computer Software Engineering, 1994, 2(3): 315-327.
149. Bach D. Bui, Marco Caccamo, Lui Sha, and Joseph Martinez. Impact of cache partitioning on multi-tasking real time embedded systems [A]. Proceedings of the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications [C], 2008, 101-110.
150. David Tam, Reza Azimi, and Michael Stumm and Livio Soares. Managing shared l2 caches on multicore systems in software [A]. Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture [C], 2007.
151. Jochen Liedtke, Hermann Hartig, and Michael Hohmuth. Os-controlled cache predictability for real-time systems [A]. Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium [C], 1997, 213-227.
152. http://sourceforge.net/projects/lpsolve/ [EB/OL], 2012

致 谢

六年多的博士生活即将结束，在这里我由衷的感谢五年来给予我无私帮助和关怀的老师、同学、朋友和家人。

首先要感谢我的导师于戈教授。在工作中，于老师头脑敏锐，治学严谨，对自己和学生的要求都非常严格。在生活中，于老师为人率真、幽默，是一位慈祥可亲的长者。您对我不但耐心地言传指导，更以您的风范与情怀深深地感动和影响着我！

感谢王义教授。您作为国际上声誉卓著的学者，不但提供给我高水平的研究平台，还对我的工作进行了大量具体指导。您那颗拳拳报国之心更是深深地激励着我。您这样的海外学子是祖国骄傲！我会以您为榜样，为祖国的尊严与荣誉奉献自己的青春！

感谢邓庆绪教授。转眼间，我与邓老师的师生情谊已有十年。您不但在工作学习上给予我大量的指导，更在生活中给予我无微不至的关怀。您的自强不息与拼搏精神，您的宽容与善良，都深深地影响着我！

感谢顾宗华老师。谢谢您在我研究工作起步阶段对我的悉心指导。您简单清净的生活方式，您的善良与从容，常常安抚我心中这个躁动的世界。

感谢软件所的鲍玉斌老师、林树宽老师、申德荣老师、王大玲老师、杨晓春老师、张伟老师、朱靖波老师、董晓梅老师、谷峪老师、胡明涵老师、寇月老师、聂铁铮老师、张俐老师、张天成老师、赵志滨老师、冯时老师、冷芳玲老师、李芳芳老师、李晓华老师、任飞亮老师、王会珍老师、吴宏林老师、张一飞老师、杨莹老师、单吉弟老师。这些可敬可爱的老师让我在软件所里感受到了家一样的温暖。

感谢实验室的同学张轶，孔繁鑫，刘炜，谷传才等等。在这里我不能一一列举你们的名字。你们的聪明才智给我的研究工作带来了取之不尽的灵感。特别感谢金曦同学，在博士论文的冲刺阶段你帮助完成我许多材料的准备工作。

特别感谢我的师兄吕鸣松。感谢这些年来，你在工作、学习和生活各方面对我的帮助。你正直的品格，兢兢业业的工作态度，一直我学习的榜样。我很幸运能遇到你这样的好兄弟。

感谢我的父母。感谢你们对我养育之恩，感谢你们为我的操劳，对我无微不至的关怀。你们的恩情我穷尽一生也无法报答，只希望儿子没有辜负你们的期望。

最后感谢我的妻子，你为了我的学业放弃了自己的工作，随我到处飘泊。这本薄薄的博士论文是献给你的礼物。

攻博期间发表的论文

1. **Nan Guan**, Mingsong Lv, Wang Yi, Ge Yu. WCET Analysis with MRU Cache: Challenging LRU for Predictability, The 18th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2012), Beijing China, April 16-19, 2012. (EI 收录: 20122415114325，第一作者)
2. **Nan Guan**, Martin Stigge, Wang Yi, Ge Yu. Parametric Utilization Bounds for Fixed-Priority Multiprocessor Scheduling, 26th IEEE International Parallel & Distributed Processing Symposium, May 21-25, 2012, Shanghai, China. (EI 检索源，第一作者)
3. **Nan Guan** and Wang Yi. Fixed-Priority Multiprocessor Scheduling: Critical Instance, Response Time and Utilization Bound, 26th IEEE International Parallel & Distributed Processing Symposium, Ph.D. Forum, May 21-25, 2012, Shanghai, China. (最佳展示论文奖) (EI 检索源，第一作者)
4. **Nan Guan**, Wang Yi, Qingxu Deng, Zonghua Gu and Ge Yu. Schedulability Analysis for Non-preemptive Fixed-Priority Multiprocessor Scheduling, Journal of Systems Architecture, 2011. (SCI 检索: 000291286900005，EI 检索: 20111913960929，第一作者)
5. **Nan Guan**, Pontus Ekberg, Martin Stigge and Wang Yi. Resource Sharing Protocols for Real-Time Task Graph Systems, The 23rd Euromicro Conference on Real-Time Systems (ECRTS'11) , July 6 - 8, 2011, Porto, Portugal. (EI 检索: 20113914359524，第一作者)
6. **Nan Guan**, Pontus Ekberg, Martin Stigge and Wang Yi. Effective and Efficient Scheduling for Certifiable Mixed Criticality Sporadic Task Systems, The 32nd IEEE Real-Time Systems Symposium (RTSS'11), November 29 - December 2, 2011, Vienna, Austria. (EI 检索: 20120614743818，第一作者)
7. **Nan Guan**, Martin Stigge, Wang Yi and Ge Yu. Fixed Priority Multiprocessor Scheduling with Liu & Layland's Utilization Bound, The 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'10) , April 12 - 15, 2010, Stockholm, Sweden. (最佳论文提名) (EI 检索: 20102613033534，第一作者)
8. **Nan Guan**, Martin Stigge, Wang Yi and Ge Yu. New Response Time Bounds of Fixed Priority Multiprocessor Scheduling, The 30th IEEE Real-Time Systems Symposium (RTSS'09), Dec 1-4, 2009 Washington, D.C., USA. (最佳论文奖) (EI 检索:

20101012758398，ISTP 检索: 000277465500036，第一作者)

9. **Nan Guan**, Martin Stigge, Wang Yi and Ge Yu. Cache-Aware Scheduling and Analysis for Multicores, The 7th International Conference on Embedded Software (EMSOFT'09), Oct. 12-16, 2009, Grenoble, France. (EI 检索: 20095212578409，第一作者)
10. **Nan Guan**, Zonghua Gu, Wang Yi and Ge Yu. Improving Scalability of Model-Checking for Minimizing Buffer Requirements of Synchronous Dataflow Graphs, The 14th Asia and South Pacific Design Automation Conference (ASP-DAC'09), Jan. 19-22, 2009, Yokohama, Japan. (最佳论文提名) (EI 检索: 20091712045009，ISTP 检索: 000265675400135，第一作者)
11. **Nan Guan**, Wang Yi, Zonghua Gu, Qingxu Deng and Ge Yu. New Schedulability Test Conditions for Non-Preemptive Scheduling on Multiprocessor Platforms, The 29th IEEE Real-Time Systems Symposium (RTSS'08), Nov. 30 - Dec. 3, 2008, Barcelona, Spain. (EI 检索: 20092612147019，ISTP 检索: 000262709900013，第一作者)
12. **Nan Guan**, Qingxu Deng, Zonghua Gu, Wenyao Xu and Ge Yu. Schedulability Analysis of Preemptive and Non-preemptive EDF on Partially Runtime Reconfigurable FPGAs, In ACM Transactions on Design Automation of Electronic Systems, 13 (4), 2008. (SCI 检索: 000259971400002，第一作者)
13. **Nan Guan**, Zonghua Gu, Mingsong Lv, Qingxu Deng and Ge Yu. Schedulability Analysis of Global Fixed-Priority or EDF Multiprocessor Scheduling with Symbolic Model-Checking, The 11th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2008), 5-7 May 2008, Orlando, Florida, USA. (EI 检索: 20083411476194，ISTP 检索: 000256430300077，第一作者)
14. **Nan Guan**, Zonghua Gu, Qingxu Deng, Shuaihong Gao and Ge Yu. Exact Schedulability Analysis for Static-Priority Global Multiprocessor Scheduling Using Model-Checking, Software Technologies for Embedded and Ubiquitous Systems, 2007. (EI 检索:20080411047636，ISTP 检索: 000252602300026，第一作者)
15. **Nan Guan**, Zonghua Gu, Qingxu Deng, Weichen Liu and Ge Yu. Improved Schedulability Analysis of EDF Scheduling on Reconfigurable Hardware Devices, Workshop on Parallel and Distributed Real-Time Systems (WPDRTS 2007). (EI 检索: 20073910825172，第一作者)
16. **Nan Guan**, Mingsong Lv, Qingxu Deng and Ge Yu. A Real-Time Scheduling Algorithm with Buffer Optimization for Embedded Signal Processing Systems. International

Symposium on Embedded Computing (SEC'07). (EI 检索: 20074210874085，第一作者)

17. Mingsong Lv, **Nan Guan**, Wang Yi and Ge Yu. McAiT - A Timing Analyzer for Multicore Real-Time Software, (tool paper), The 19th International Symposium on Automated Technology for Verification and Analysis (ATVA 2011), Taipei, Taiwan, 11 - 14 October 2011. (EI 检索: 20114214439207，第二作者)
18. Xi Jin, **Nan Guan**, Qingxu Deng and Wang Yi. Memory Aware Mapping for Network-on-Chips, The 17th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2011), August 28-31, Toyama, Japan. (EI 检索: 20120214674858，第二作者)
19. Martin Stigge, Pontus Ekberg, **Nan Guan** and Wang Yi. On the Tractability of Digraph-Based Task Model, The 23rd Euromicro Conference on Real-Time Systems (ECRTS'11) , July 6 - 8, 2011, Porto, Portugal. (EI 检索: 20113914359539，第三作者)
20. Yi Zhang, **Nan Guan**, Wang Yi and Yanbin Xiao. Implementation and Empirical Comparison of Partitioning-based Multi-core Scheduling, The 6th IEEE International Symposium on Industrial Embedded Systems (SIES'11), June 15th - 17th, 2011, Vasteras, Sweden. (EI 检索: 20113514266470，第二作者)
21. Martin Stigge, Pontus Ekberg, **Nan Guan** and Wang Yi. The Digraph Real-Time Task Model, The 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'11) , April 11 - 14, 2011, Chicago, US. (最佳论文提名) (EI 检索: 20112214019148，ISTP 检索: 000299168100007，第三作者)
22. Yi Zhang, **Nan Guan** and Wang Yi. Towards the Implementation and Evaluation of Semi-Partitioned Multi-Core Scheduling. PPES 2011: 42-46. (第二作者)
23. Fanxin Kong, **Nan Guan**, Qingxu Deng and Wang Yi. Energy-Efficient Scheduling for Parallel Real-Time Tasks Based on Level-Packing, The 26th Symposium On Applied Computing (SAC'11), March 21-24, 2011, TaiChung, Taiwan. (EI 检索: 20112514080668，第二作者)
24. Mingsong Lv, Wang Yi, **Nan Guan** and Ge Yu. Combining Abstract Interpretation with Model Checking for Timing Analysis of Multicore Software, The 31st IEEE Real-Time Systems Symposium (RTSS'10) November 30 - December 3, 2010 San Diego, CA, USA. (最佳论文提名) (EI 检索: 20110813691757，ISTP 检索: 000287965300031，第三作者)
25. Mingsong Lv, **Nan Guan**, Qingxu Deng, Ge Yu and Wang Yi. Static worst-case execution time analysis of the μC/OS-II real-time kernel, Frontiers of Computer Science

in China 4(1): 17-27 (2010). (SCI 检索: 000292503700002，EI 检索: 20100812724380，第二作者)

26. Xi Jin, **Nan Guan**, Mingsong Lv and Qingxu Deng. Improving the Performance of Shared Memory Communication in Impulse C, IEEE Embedded Systems Letters, Volume: PP Issue: 99. (EI 检索: 20103913253643，第二作者)
27. Mingsong Lv, **Nan Guan**, Yi Zhang, Rui Chen, Qingxu Deng, Ge Yu and Wang Yi. WCET Analysis of the uC/OS-II Real-Time Kernel, the 7th IEEE/IFIP International Conference on Embedded and Ubiquitous Computing (EUC 2009). (EI 检索: 20094912529061，第二作者)
28. Mingsong Lv, **Nan Guan**, Yi Zhang, Qingxu Deng, Ge Yu and Jianming Zhang. A Survey of WCET Analysis of Real-Time Operating Systems, the 6th IEEE International Conference on Embedded Software and Systems (ICESS 2009). (EI 检索: 20094212372971，ISTP 检索: 000271940700009，第二作者)
29. Mingsong Lv, Zonghua Gu, **Nan Guan**, Qingxu Deng and Ge Yu. Performance Comparison of Techniques on Static Path Analysis of WCET, the 6th IEEE/IFIP International Conference on Embedded and Ubiquitous Computing (EUC 2008). (EI 检索: 20091512017854，ISTP 检索: 000264238700014，第三作者)
30. Qingxu Deng, Fanxin Kong, **Nan Guan**, Mingsong Lv and Wang Yi. On-line Placement of Real-Time Tasks on 2D Partially Run-time Reconfigurable FPGAs, the 5th IEEE International Symposium on Embedded Computing (SEC 2008). (ISTP 检索: 000263713900004，第三作者)
31. Mingsong Lv, Qingxu Deng, **Nan Guan**, Yaming Xie and Ge Yu. ARMISS: An Instruction Set Simulator for the ARM Architecture. ICESS 2008: 548-555. (EI 检索: 20083911587221，ISTP 检索: 000259300200079，第三作者)
32. Zonghua Gu, Mingxuan Yuan, **Nan Guan**, Mingsong Lv, Qingxu Deng and Ge Yu. Static Scheduling and Software Synthesis of Dataflow Models with Symbolic Model-Checking, The 28th IEEE Real-Time Systems Symposium (RTSS'07), Tucson, Arizona, December 2007. (EI 检索: 20083211446330，ISTP 检索: 000253708400033，第三作者)

攻博期间参与的项目

1. 国家 863 项目（2007AA01Z181），面向可重构计算系统的实时调度问题与操作系统技术的研究，2007 年 8 月~2009 年 12 月，主要研究人员。
2. 国家自然科学基金面上项目（60973017），多核系统中实时调度策略的设计与分析技术的研究，2010 年 1 月~至今，执行负责人。
3. 国家自然科学基金青年项目（61100023），多核实时系统中共享资源管理与分析技术研究，2012 年 1 月~至今，主要研究人员。
4. 教育部中央高校基本科研业务费（N100204001），支持 MPSoC 体系结构设计空间探索及软件开发的高性能虚拟平台，2011 年 1 月~至今，主要研究人员。
5. 教育部中央高校基本科研业务费（N100304001），多核实时系统最坏情况执行时间分析技术研究及分析软件设计，2011 年 1 月~至今，主要研究人员。
6. 教育部科技创新工程重大培育计划（706016），面向智能化装备的嵌入式平台开发及应用示范，2006 年 8 月~2009 年 12 月，主要研究人员。
7. 辽宁省自然科学基金（20082032），基于模型检测的片上多处理器系统实时性分析技术的研究，2008 年 5 月~2009 年 5 月，主要研究人员。

作者简介

关楠，男，满族，1981 年出生于辽宁沈阳。1999 年考入东北大学计算机科学与技术专业，2003 年毕业，获学士学位。同年，保送本校研究生，从师邓庆绪教授，从事实时嵌入式系统方面的研究工作，2006 年毕业，获硕士学位。同年考入东北大学研究生院，攻读嵌入式系统及应用专业博士学位，从师于戈教授，主要从事实时系统中面向多核系统的实时调度算法方面的研究工作。

2006 年起，一直致力于实时系统中面向多核系统实时调度算法的研究工作。研究成果发表（已录用）在包括《ACM Transactions on Design Automation of Electronic Systems》国际期刊、《RTSS》国际会议、《RTAS》国际会议等，其中于 2009 年以第一作者身份获得《RTSS》国际会议最佳论文奖，并多次获得《IPDPS》、《RTSS》、《RTAS》等国际会议最佳展示论文奖和最佳论文提名。博士阶段共发表论文 32 篇，其中第一作者被 SCI 收录 2 篇，第一作者被 EI 收录 13 篇。

作为项目执行负责人参与了参与了国家自然科学基金项目“多核系统中实时调度策略的设计与分析技术的研究（60973017）”。作为主要参与人已完成和正在参与的科研项目主要有国家 863 项目（2007AA01Z181）“面向可重构计算系统的实时调度问题与操作系统技术的研究”，教育部科技创新工程重大培育计划（706016）“面向智能化装备的嵌入式平台开发及应用示范”，教育部中央高校基本科研业务费（N100304001）“多核实时系统最坏情况执行时间分析技术研究及分析软件设计”， 教育部中央高校基本科研业务费（N100204001），支持 MPSoC 体系结构设计空间探索及软件开发的高性能虚拟平台，辽宁省自然科学基金（20082032）“基于模型检测的片上多处理器系统实时性分析技术的研究”等。